Instituto de Computação
Universidade Estadual de Campinas

Estudo de algoritmo evolutivo com codificação real na geração de dados de teste estrutural e implementação de protótipo de ferramenta de apoio

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por André Vinicius Buzzo e aprovada pela Banca Examinadora.

Campinas, 17 de Fevereiro de 2011.

Eliane Martins

Eliane Martins (Orientadora)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Fabiana Bezerra Müller – CRB8 / 6162

Buzzo, André Vinicius
B989e Estudo de algoritmo evolutivo com codificação real na geração de dados de teste estrutural e implementação de protótipo de ferramenta de apoio/André Vinicius Buzzo-- Campinas, [S.P. : s.n.], 2011.

Orientador : Eliane Martins.
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1.Computação evolutiva. 2.Algoritmos genéticos. 3.Software - Testes. 4.Engenharia de software. 5.Otimização extrema generalizada. I. Martins, Eliane. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Study of real-coded evolutionary algorithm to test data generation and implementation of prototype tool

Palavras-chave em inglês (Keywords): 1.Evolutionary computation. 2.Genetic algorithms. 3.Software testing. 4.Software engineering. 5.Generalized extremal optimization.

Área de concentração: Ciência da Computação

Titulação: Mestre em Ciência da Computação

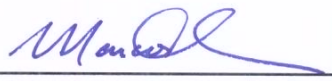
Banca examinadora: Profa. Dra. Eliane Martins (IC – UNICAMP)
Prof. Dr. Eduardo Candido Xavier (IC – UNICAMP)
Prof. Dr. Marcos Lordello Chaim (EACH - USP)

Data da defesa: 17/02/2011

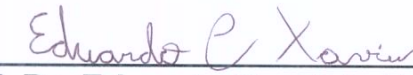
Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

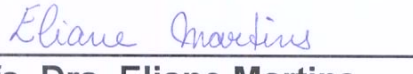
Dissertação Defendida e Aprovada em 17 de fevereiro de 2011, pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Marcos Lordello Chaim
EACH / USP



Prof. Dr. Eduardo Candido Xavier
IC / UNICAMP



Profa. Dra. Eliane Martins
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Estudo de algoritmo evolutivo com codificação real na geração de dados de teste estrutural e implementação de protótipo de ferramenta de apoio

André Vinicius Buzzo

Fevereiro de 2011

Banca Examinadora:

- Eliane Martins (Orientadora)
- Eduardo Candido Xavier
  Instituto de Computação
  Universidade Estadual de Campinas
- Marcos Lordello Chaim
  Escola de Artes, Ciências e Humanidades
  Universidade de São Paulo
- Ariadne Maria B. Rizzoni Carvalho (Suplente)
  Instituto de Computação
  Universidade Estadual de Campinas
- Mario Jino (Suplente)
  Faculdade de Engenharia Elétrica e de Computação
  Universidade Estadual de Campinas

Resumo

A geração automática de dados de teste pode ser abordada como um problema de otimização e algoritmos evolutivos se tornaram um foco de muita pesquisa nesta área. Recentemente um novo tipo de algoritmo evolutivo chamado GEO (GEO – *Generalized Extremal Optimization*) tem sido explorado em uma grande classe de problemas de otimização. Neste trabalho é apresentado o uso do algoritmo evolutivo GEO com codificação real - GEOreal - na geração de dados de teste. O desempenho deste algoritmo é comparado com diversos outros algoritmos e para melhor avaliar os resultados, duas funções objetivo - que mapeiam o problema de geração de dados em um problema de otimização - foram utilizadas. O algoritmo GEOreal combinado com a função objetivo Bueno e Jino obtiveram os melhores resultados nos problemas abordados. Um protótipo foi desenvolvido implementando todos os conceitos envolvidos neste trabalho e o seu desempenho foi comparado com outras ferramentas já disponíveis no mercado. Os resultados mostraram que este protótipo superou as ferramentas comparadas ao minimizar o tempo dispendido no esforço de gerar os dados de teste.

Abstract

Automatic test data generation can be approached as an optimization problem and evolutionary algorithms have become a focus of much research in this area. Recently a new type of evolutionary algorithm called GEO (GEO - *Generalized Extremal Optimization*) has been explored in a large class of optimization problems. This paper presents the use of evolutionary algorithm with real coding GEO - GEOreal - in test data generation. The performance of this algorithm is compared with several other algorithms and to better compare the results two objective functions - that map the problem of generating data in an optimization problem - were used. The algorithm GEOreal combined with the function Bueno and Jino had the best results in the problems addressed. A prototype was developed implementing all the concepts involved in this work and its performance was compared with other tools already available. The results showed that this prototype was better than the compared tools when minimizing the time spent in the effort to test data generation.

Agradecimentos

Agradeço, em primeiro lugar, a meu Pai que esta no céu. Ele me sustentou durante estes três anos de estudo dando saúde e motivação para suportar - ao mesmo tempo - trabalho, família e mestrado.

Não menos importante foi a ajuda da minha esposa Fernanda que não deixou de me auxiliar durante toda esta caminhada. Amo você! Que Deus permita a sua companhia durante toda a nossa vida.

Agradeço também aos meus excelentes orientadores Eliane Martins e Fabiano Sousa. Sem eles não conseguiria desenvolver este enriquecedor aprendizado profissional e pessoal. Graças a sua dedicação e apoio pude encarar este desafio e superá-lo.

Por fim não poderia deixar de agradecer meus pais, meus sogros e outros familiares por todo apoio que me deram, principalmente cuidando de nossa filha Luiza enquanto eu dedicava horas no desenvolvimento deste trabalho.

Conteúdo

Glossário

CUT Class Under Test. 54

FO Função Objetivo. 5

GEO Generalized Extremal Optimization. vii

JGAP Java Generic Algorithms Package. 41

MUT Method Under Test. 56

RCGA Real Coded Genetic Algorithm. 15

SGA Simple Genetic Algorithm. 14

SOC Self-Organized Criticality. 16

SUT Software Under Test. 4

Capítulo 1

Introdução

1.1 Contexto

O mundo corporativo atual tem exigido um alto nível de qualidade no desenvolvimento de novas soluções baseadas em *software*. A confiabilidade e o preço estão diretamente ligados aos processos utilizados para a criação destes produtos. Como o nível de complexidade e o tamanho dos sistemas envolvidos tem crescido nas últimas décadas, tornou-se necessário a utilização de uma grande equipe de profissionais além de novos processos para o desenvolvimento de um novo produto. Um exemplo de processo que prevê a utilização de uma grande quantidade de recursos (pessoas e tempo) no desenvolvimento de um *software* é o *Rational Unified Process* [1]. Nesta metodologia a garantia de qualidade final é feita através de ciclos de testes para garantir a integridade do sistema como um todo.

Segundo Pressman [56] não é incomum as empresas investirem cada vez mais os esforços do projeto em testes. Testar um *software* ajuda na identificação de erros e um benefício direto é que isto demonstra que as funções de *software* estão aparentemente de acordo com as especificações. O processo de teste pode se tornar complexo se critérios para montar o conjunto de dados de entrada do caso de teste não forem escolhidos corretamente. Ainda assim, todo *software* produzido por uma empresa precisa ser testado para garantir um mínimo de qualidade. Um passo muito importante na atividade de testes é avaliar o quanto um conjunto de dados de teste exercitam o *software*. Partindo desta ideia, diversos critérios foram definidos para auxiliar esta avaliação. Após escolher um ou mais critérios de teste é necessário encontrar dados de teste que atendam aos mesmos. Dados de teste exaustivos executando em todo o domínio de entrada são impraticáveis [14]. Tendo em vista isto, é importante utilizar algum algoritmo para automatizar a geração de dados de teste [2]. É comum que estes algoritmos sejam baseados em geração

[1] http://www-01.ibm.com/software/awdtools/rup/

aleatória de dados ou com foco na estrutura/funcionalidade do trecho de *software* a ser testado.

Infelizmente não é possível abordar a geração de dados de teste de forma determinística utilizando, por exemplo, programação linear [40]. Esses algoritmos não são aplicáveis em problemas reais pois não podem ser representados por equações lineares, e além disso, frequentemente existem restrições e funções complexas. Apesar disso a geração automática de dados de teste possui características que permitem tratá-la como um problema de otimização. É possível explorar os domínios de entrada do *software* em teste (SUT – *Software Under Test*) utilizando alguma metaheurística para gerar os dados. Entre estas metaheurísticas estão os Algoritmos genéticos [61, 34, 14, 50] que foram apresentados pela primeira vez por Holland [22]. Estes algoritmos seguem os princípios da evolução proposto por Charles Darwin buscando encontrar uma solução ótima para um problema. Recentemente um novo tipo de algoritmo evolutivo chamado GEO (GEO – *Generalized Extremal Optimization*) [35, 36] inspirado no modelo evolucionário de Bak-Sneppen [47] tem sido explorado em uma grande classe de problemas de otimização inclusive para a geração de dados de teste. Dentre suas implementações abordando este tipo de problema temos o algoritmo GEOreal que trabalha internamente com população na representação real.

1.2 Caracterização do Problema

Apesar do aparecimento de novas técnicas para o controle dos processos durante o desenvolvimento de uma solução baseada em *software*, ainda grande parte do desenvolvimento do produto é feita através de trabalho humano. Com isso há grande chance do *software* apresentar falhas e erros. Por isso as empresas ainda empregam boa parte de seus esforços em testes do produto [26].

A automação da geração de dados de teste tem sido uma saída muito explorada nas últimas décadas [40, 11, 6, 49]. Difundiu-se o estudo de técnicas visando este tipo de geração de dados com o objetivo de encontrar entradas que satisfaçam um determinado critério. No entanto, a geração automática de dados para os SUT tem limitações pois em muitos casos o problema é indecidível [14]: um exemplo é o uso do critério de caminhos que visa cobrir todo caminho de um SUT. Sabe-se que isso é impossível pois a existência de *loops* podem levar a um número de caminhos infinitos.

A utilização de dados aleatórios não é promissora por não explorar completamente os comportamentos do *software*. Desta forma a utilização de metaheurísticas para a geração dos dados de entrada para SUT tem sido uma área muito explorada nos traba-lhos científicos dos últimos anos [59, 10, 39, 53]. Dentre as várias frentes de estudo está a utilização de algoritmos genéticos [34, 14, 50] e suas variações. Estas técnicas necessitam

de uma função objetivo que guie a busca no espaço de domínio das variáveis de entrada do SUT. A escolha desta função impacta profundamente o desempenho da busca pela solução do problema. Portanto é importante pesquisar e encontrar novas opções de funções objetivos que possam se adaptar melhor a cada tipo de problema.

Entre os algoritmo evolutivos, uma das técnicas recentes é o GEO (e suas implementações). Esta metaheurística tem demonstrado ser simples por possuir apenas uma variável de ajuste para configurar seu algoritmo ao problema proposto.

1.3 Objetivo

O objetivo desta pesquisa é dar continuidade ao trabalho de Abreu [58] que estudou a utilização do GEO para a geração automática de dados de teste. A principal contribuição deste trabalho foi o estudo de uma nova implementação do algoritmo GEO que utiliza a codificação real para representar sua população. Esta implementação impactou positivamente no desempenho deste algoritmo ao abordar os problemas propostos. Além disto, novos algoritmos foram implementados para a comparação dos dados coletados e para todos os problemas abordados o gráfico da avaliação da função objetivo foi plotada. Isto permitiu uma melhor interpretação dos resultados.

Outra contribuição deste trabalho foi a implementação de outra função objetivo além da já conhecida *Similarity* [13]: a função objetivo Bueno e Jino [48]. Ambas foram implementadas e utilizadas para guiar as metaheurísticas no domínio de busca. Desta forma foi verificada a influência da função objetivo na eficácia da busca. O resultados mostraram um melhor desempenho dos algoritmos quando guiados pela FO (FO - função objetivo)Bueno e Jino em todos os problemas atacados.

Por fim este trabalho permitiu o desenvolvimento de um protótipo, nomeado JET, que aplicou diversos conceitos estudados neste trabalho. Os algoritmos GEO e as funções objetivos foram implementadas no JET e este foi comparado com outras ferramentas com o mesmo propósito disponíveis na internet. Os resultados mostraram que o protótipo tem potencial para ser evoluído em uma ferramenta pois obteve bom desempenho na geração de dados de teste.

1.4 Estrutura do Texto

Esta seção apresentou o contexto, as motivações e os objetivos deste trabalho. A próxima descreve o conceito de testes evolutivos[2] apresentando um resumo sobre testes, algorit-

[2]O uso desta terminologia é usada aqui pois o algoritmo abordado neste trabalho é um algoritmo evolutivo (GEO).

mos evolutivos, o novo algoritmo GEO e sua variação o GEOreal. A seção 3 expõe os trabalhos existentes que abordam os assuntos desta pesquisa. Na seção 4 são mostrados os experimentos realizados e os resultados obtidos. A seção 5 expõe o protótipo JET e na última seção é apresentado a conclusão deste trabalho.

Capítulo 2

Testes Evolutivos

2.1 Testes

Verificação e Validação (V & V) são processos que são empregados no desenvolvimento de um *software*. O primeiro diz respeito às atividades que certificam que um *software* implementa corretamente uma determinada função. Já a validação refere-se a atividades que confirmam se o *software* atende os requisitos do usuário. Testes são um subconjunto desses processos e tem como objetivo revelar falhas do SUT, em especial aqueles que acabem por impedir que requisitos do sistema sejam atendidos [25]. Por este motivo é importante que a fase de testes seja planejada antes mesmo da implementação do sistema pois é nesta fase (de planejamento e análise) no qual os documentos de requisitos estão disponíveis.

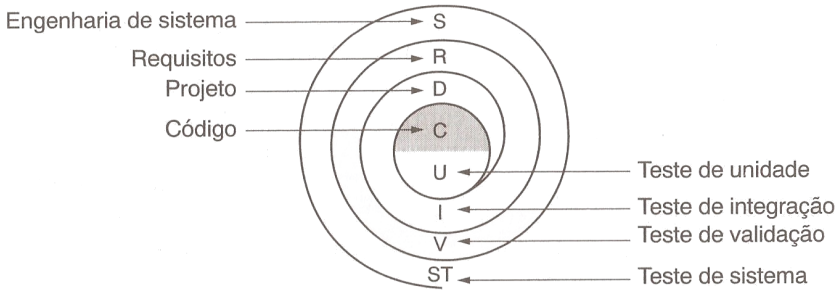


Figura 2.1: Espiral de teste

O processo de desenvolvimento de um *software* pode seguir a estratégia chamada "espiral de teste" 2.1 apresentada por Pressman [56]. Esta estratégia permite a execução de testes de baixo nível (sobre o código) e alto nível (funcionalidades principais do *software*).

Inicialmente são feitos testes nos módulos individuais do projeto garantindo que cada um deles funcione corretamente. Este processo pode ser feito em paralelo e é chamado de teste de unidade pois foca em trechos do código-fonte do *software*. Caso não seja possível ter acesso ao código-fonte, é possível verificar por falhas nas interfaces do módulo explorando, por exemplo, por entradas maiores ou menores do que as esperadas.

Na iteração seguinte temos testes de integração em que os pequenos módulos são combinados para se chegar a um ou mais pacotes de *software*. Aqui a estrutura do programa é montada realizando-se testes nas interfaces para garantir que a entrada e saída dos dados estejam consistentes. Assim que o sistema foi integrado é possível lançar mão de testes de mais alto nível como os de validação que são usados para verificar que o *software* responde aos critérios comportamentais e funcionais exigidos. Por fim, o teste de sistema confere se o *software* combinado com os outros elementos do sistema comportam-se conforme o modelo esperado [56].

Durante as fases de testes faz-se uso de muitas técnicas. Durante o passo de teste de unidade geralmente usa-se a técnica conhecida como caixa branca [46]. O acesso direto ao código fonte permite exercitar caminhos de controle e de estrutura. No teste de integração é mais comum o uso da técnica de caixa preta [20] já que o foco aqui é a construção da arquitetura do *software*. Ainda assim é possível o uso da técnica de caixa branca para a validação das estruturas de controle [56]. Nos testes de validação a técnica de caixa preta é a mais utilizada [56]. Aqui já não se faz mais acesso a qualquer código-fonte mas apenas estruturas do modelo. Por último, na etapa chamada teste de sistema, são executados testes de stress e desempenho.

Como neste trabalho será abordado apenas a técnica caixa branca, esta é detalhada na próxima seção.

2.1.1 Testes de Caixa Branca

Os testes de caixa branca são aqueles em que se tem acesso à implementação e à estrutura do *software*. A partir destas informações é possível derivar conjuntos de testes de forma a exercitar estas estruturas. Por esse motivo estes tipos de testes geralmente são aplicados a pequenos módulos ou funções do *software*.

Uma forma de representar a estrutura do módulo para que o projetista possa derivar testes é através de grafos de fluxo. Estes grafos direcionais permitem uma visualização estrutural da lógica do trecho de código. Um exemplo de CFG (CFG - *Control Flow Graph*) é mostrado na figura 2.2.

Cada nó representa um bloco básico, ou seja, um trecho de código sem condicionais ou *loops*. As arestas representam os caminhos possíveis para a execução do desvio de fluxo. O grafo sempre possui pelo menos dois nós: o bloco de início e o bloco de saída. O

primeiro representa o início do programa e o segundo o fim do fluxo deste. Existem nós com mais de uma aresta de saída. Estes são blocos chamados nós condicionais.

Um critério de teste conhecido na técnica de caixa branca é a criação de testes que exercitem caminhos de um SUT. Um caminho é uma sequência de nós de um CFG desde o nó de início até o de saída. Um SUT pode possuir inúmeros caminhos e cobri-los pode ser computacionalmente impraticável: o número de caminhos de um programa é exponencial ao número de predicados [62] e a presença de *loops* pode levar a um número infinito de caminhos. Além disso podem existir caminhos não-executáveis, ou seja, caminhos que não serão executados com nenhum dado de teste. Por este motivo este critério geralmente envolve a seleção de um conjunto de caminhos do SUT [13].

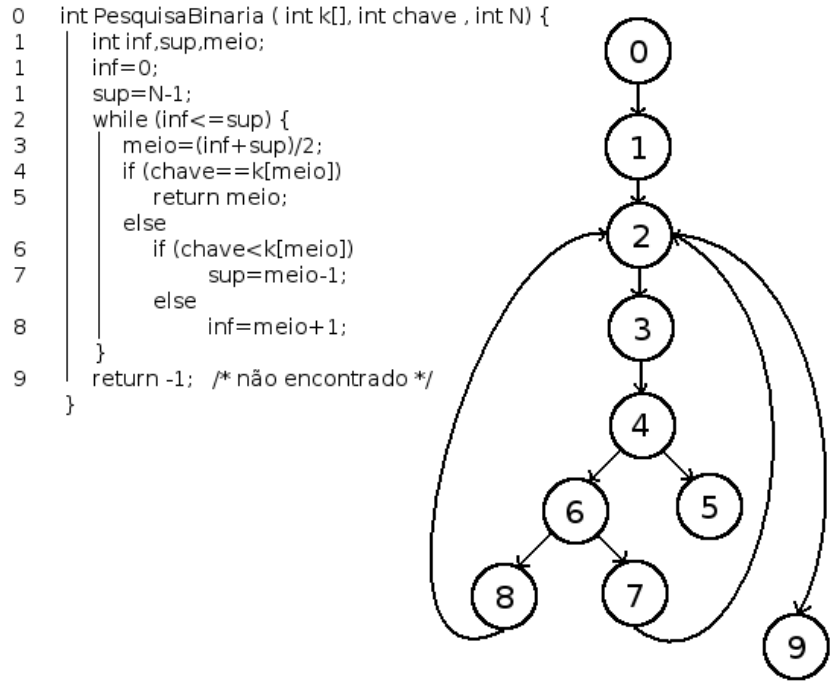


Figura 2.2: CFG do algoritmo Busca Binária

Existe uma métrica de *software* chamada complexidade ciclomática. Esta, quando aplicada no contexto de teste de caminho, determina quantos caminhos independentes possui um CFG de um programa. Este valor pode ser calculado de algumas maneiras:

1. V(G) = E - N + 2, sendo V(G) a complexidade ciclomática, G é o fluxo de grafo, E é o número de ramos e N o número de nós. Para nosso exemplo na figura 2.2 temos V(G) = 11 - 9 + 2 = 4.

2. V(G) = P + 1, onde P é o número de nós condicionais. Para a figura 2.2, V(G) = 3 + 1 = 4.

3. V(G) = número de regiões (R) de um grafo que podem ser contabilizadas através da fórmula de Euler [45]: N - E + R = 2. Esta fórmula é equivalente à R = E - N + 2, mesma descrita no primeiro item. Para a figura 2.2 R = V(G) = 4.

Um caminho independente é qualquer caminho que introduza pelo menos um novo conjunto de instruções ou uma nova condição. V(G) oferece um limite máximo para o número de caminhos independentes de um CFG. Da figura 2.2 pode-se escolher os seguintes caminhos independentes (serão quatro pois V(G) = 4):

1. 0-1-2-9

2. 0-1-2-3-4-5

3. 0-1-2-3-4-6-8...

4. 0-1-2-3-4-6-7...

Existem outros critérios que podem ser utilizados para derivar testes a partir da técnica caixa branca. Um deles é o de testar todos os predicados - testes seriam derivados com o objetivo de verificar todas as possibilidades de saída de todos as decisões lógicas (testar tanto o falso quanto o verdadeiro). Este critério acabaria por testar o equivalente a todas as arestas do CFG. Há também o critério de instruções: todas as instruções do código-fonte devem ser executadas pelo menos uma vez o que se traduziria, observando a figura 2.2, em todos os nós do CFG.

2.1.2 Automatização da Geração de Dados de Teste

Com o intuito de diminuir o tempo, o risco, aumentar a confiabilidade e a produtividade, ferramentas de automação da execução de testes de *software* podem ser utilizadas. Para isto é importante conhecer o conceito de critério de teste. Este avalia a qualidade dos dados de teste gerados sendo que esta qualidade está diretamente ligada à satisfação dos requisitos de teste escolhidos pelo usuário. Portanto satisfazer este requisito implica na existência de um dado de entrada que exercite funções ou instruções do *software* deste requisito.

Quando um determinado requisito de teste não é satisfeito por um dado de entrada, este pode ser modificado até que o requisito seja satisfeito. Este esforço pode ser feito manualmente mas acaba por dispender grandes recursos do projeto. Por este motivo a automação da geração de dados de teste se mostra interessante. Dentre as principais abordagens de geração automática de dados de teste temos a aleatória, a execução simbólica

e a execução dinâmica.

A geração de dados de teste aleatória simplesmente seleciona valores do domínio de entrada e os executa no SUT. Entre as suas vantagens está o baixo custo [62] mas o fato desta abordagem apresentar dificuldade em exercitar determinados condicionais, por exemplo de igualdade entre dois valores, a torna ineficiente [14]. O mesmo seria observado em um SUT com um grande domínio de entrada na qual poucos valores (ou mesmo apenas um) satisfizessem um determinado requisito de teste. Isso implica em baixíssima probabilidade de satisfazer o requisito. Myers [21], um grande pesquisador desta área, chega a afirmar que esta é a pior abordagem de geração de dados de teste.

A geração de dados através da execução simbólica foi primeiramente proposta por *King* na década de 70 [12]. A ideia é obter uma caracterização matemática do *software* através da atribuição de variáveis simbólicas às variáveis do programa. Desta maneira uma função matemática é capaz de representar os caminhos funcionais do *software* ao atribuir nomes simbólicos às variáveis de entrada [23]. Para este cálculo existe um passo chamado de geração das expressões simbólicas. Estas expressões representam os requisitos para executar um determinado caminho ou predicado. Desta maneira obtém-se um conjunto de restrições de igualdade e desigualdade utilizando as variáveis de entrada. Estas restrições podem ser lineares ou não lineares e definem um subconjunto do domínio de entrada que exercitam o caminho escolhido.

Apesar disto a execução simbólica possui desvantagens [14]. A manipulação das expressões simbólicas é um processo computacional dispendioso e apresenta problemas com programas que possuem execução de variáveis que dependem de vetores, *loops* e chamadas de funções [14, 6]. Em problemas reais estes são os principais recursos computacionais usados o que dificulta a utilização desta abordagem apesar de, em teoria [14], a execução simbólica poder ser utilizada.

A execução dinâmica é baseada na execução real do programa e análise dinâmica de fluxo de dados. De forma prática, dados são gerados e executados no programa. Em seguida esses são avaliados e melhorados utilizando-se métodos de otimização de funções para que então satisfaçam um determinado requisito. Estes métodos de otimização conseguem extrair informações da execução do SUT, por exemplo, através de uma instrumentação, e assim determinar se o requisito de teste foi satisfeito.

Diversos autores [48, 14, 50, 23, 43, 7] utilizam a abordagem dinâmica em virtude de suas vantagens em relação às outras abordagens. O conceito mais trabalhado é a transformação da geração de dados em um problema de otimização, geralmente representado por uma função matemática que deve ser maximizada ou minimizada. Todavia dependendo da função matemática (função objetivo) e da técnica de otimização utilizada não se garante que a solução será encontrada. Um exemplo é o caso em que a função objetivo

possui diversos máximos locais e apenas um máximo global e dependendo do método de otimização utilizado este pode ficar preso em um solução que é máximo local e não encontrar o máximo global.

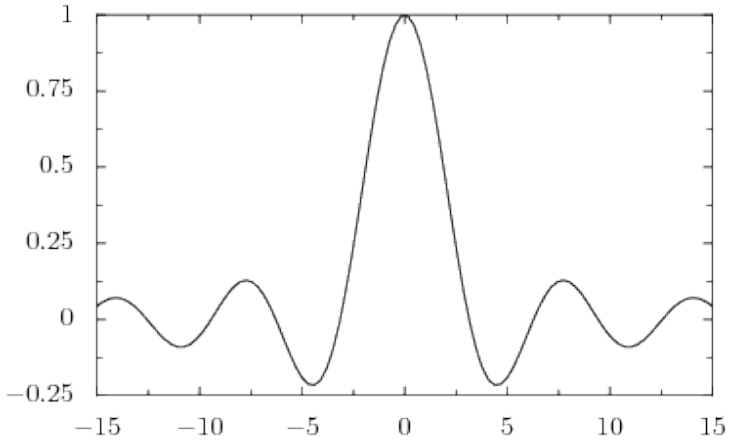


Figura 2.3: Função com máximos e mínimos locais e globais

$$f(x) = \begin{cases} 1 & \text{x} = 0 \\ \frac{sin(x)}{x} & \text{x} \neq 0 \end{cases} \tag{2.1}$$

Como exemplo pode-se observar a figura 2.3 que representa a função 2.1. Ela possui diversos máximos e mínimos locais e um máximo global localizado em $x = 0$. Esta função pode representar um requisito de teste que deve ser satisfeito e isto será alcançado quando $x = 0$. Um método de otimização conhecido e que pode ser exemplificado aqui é o *Hill Climbing* [41]. Este algoritmo inicia aleatoriamente em algum ponto do domínio e em seguida examina sua vizinhança procurando por um ponto com maior valor de avaliação $f(x)$. Caso encontre, este ponto é transformando no ponto corrente e o processo é repetido. O algoritmo continua até encontrar um ponto onde sua vizinhança não possui um valor de avaliação $f(x)$ maior e assim termina retornando x. A solução encontrada pode ser um máximo local ou global o que torna este algoritmo passível de não encontrar o x desejado. Uma maneira de solucionar esta restrição é reiniciar o algoritmo diversas vezes e armazenar o ponto onde ocorreu a maior avaliação [40]. Isto mostra que este tipo de algoritmo é altamente dependente da solução inicial.

Pensando em minimizar este problema um tipo de abordagem muito utilizada atualmente é o uso de algoritmos baseados em metaheurística que realizam uma busca global (exemplos de métodos de busca global podem ser visualizadas na figura 2.4). Entre estes estão os Algoritmos Evolutivos que serão descritos na próxima seção.

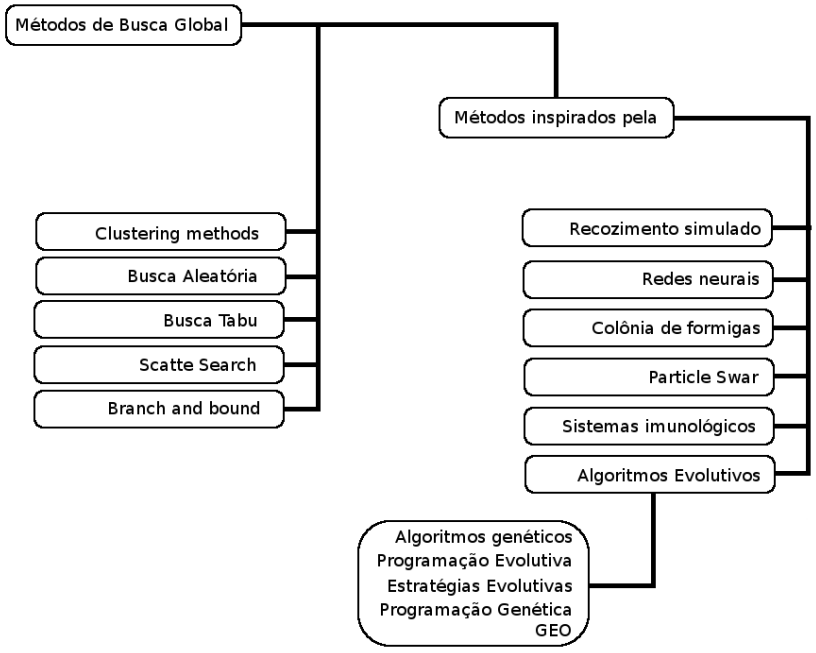


Figura 2.4: Métodos de busca global

2.2 Algoritmos Evolutivos

Algoritmos evolutivos, ou computação evolutiva, constituem um conjunto de metaheuristicas baseados nos conceitos de evolução natural e genética [18, 65]. Uma de suas maiores aplicações é como método de otimização global. Dentre os subgrupos dos algoritmos evolutivos estudados hoje na literatura [18] existem as Estratégias Evolutivas (EE), Programação Evolutiva (PE), Programação Genética (PG) e os Algoritmos Genéticos (AG). Todos compartilham de características comuns: evoluir uma população em direção a um objetivo enquanto os indivíduos sofrem seleção, recombinação e mutação. Estas estratégias favorecem espécies que estão melhores adaptadas ao ambiente de forma que venham a sobreviver e continuar a evoluir. De forma geral o modelo de um algoritmo evolutivo pode ser descrito pelo algoritmo 2.5.

Um algoritmo evolutivo mantém uma População(t) = $\{x^{t}{}_{1} \ldots x^{t}{}_{n}\}$ de indivíduos por geração (ou iteração). Cada indivíduo é uma potencial solução para o problema trabalhado sendo que sua estrutura e codificação são dependentes do tipo de algoritmo evolutivo implementado. Cada indivíduo possui uma medida de adaptação indicando sua potencialidade para solucionar o problema. Através de diversas iterações, que são terminadas a partir de um critério de parada – solução encontrada, número de iterações máxima atingida, etc. – uma nova população é criada ao se executar diversos "operadores genéticos" sobre a antiga população. Esses operadores incluem:

```
início
        t = 0
        init Pop(t)
        avalie Pop(t)
        enquanto (condição de parada não satisfeita) faça
        início
                t = t + 1
                selecione Pop(t) a partir de Pop(t-1)
                altere Pop(t)
                avalie Pop(t)
        fim
fim
```

Figura 2.5: Estrutura de um AE [66]

1. seleção: através de um mecanismo alguns indivíduos são selecionados (pode-se privilegiar os mais adaptados).

2. alteração: pode ser uma modificação pequena em um atributo do indivíduo (mutação) ou através da combinação de dois ou mais indivíduos (recombinação).

Apesar dos algoritmos evolutivos compartilharem diversas características, existem aspectos nos quais diferem: tipo de codificação do indivíduo, como são selecionados os indivíduos, a ordem e quais operadores genéticos são empregados e como a população inicial é criada.

Ao se trabalhar com AEs diversas opções devem ser tomadas antes de se executar a busca pela solução do problema abordado. Deve-se escolher qual algoritmo será usado, qual será a representação, operadores e probabilidades do algoritmo. Diversos autores [19, 65] têm mostrado que o valor destes parâmetros impactam o desempenho do algoritmo drasticamente. Alguns desses parâmetros são inerentes à escolha do próprio algoritmo como por exemplo a representação do indivíduo ou os operadores utilizados. Mas existem outros parâmetros como probabilidade de mutação, de recombinação, tamanho da população, que precisam ser "adaptados" ao problema. Ou seja, de alguma maneira diversos valores destes parâmetros precisam ser testados e avaliados a partir do problema para se encontrar aqueles que se encaixem satisfatoriamente.

2.2.1 Algoritmos Genéticos

A forma mais simples de implementação dos chamados Algoritmos Genéticos é o SGA (SGA - *Simple Genetic Algorithm*) que foi proposta por Holland [22] e consiste em uma população iniciada aleatoriamente de indivíduos em que cada um representa uma solução para o problema. Em seguida toda a população é avaliada e pais são selecionados conforme sua adaptação para serem combinados para produzirem novos filhos. Estes últimos passam por um processo de baixa probabilidade de mutação sendo em seguida reinseridos

na população. A partir daqui o ciclo do algoritmo genético entra em repetição até que algum critério de parada (tempo, número de iterações da FO, etc.) seja alcançado.

Nos algoritmos genéticos a recombinação de indivíduos chamados pais e a aplicação do operador mutação criam novos indivíduos chamados de filhos que podem substituir a antiga população (pais) ou serem integrados a esta. Este ciclo tende a levar a população, em cada nova iteração, a uma melhor adaptação ao ambiente (problema). Desta maneira os algoritmos genéticos são uma classe de algoritmos com bom aproveitamento de melhores soluções ao mesmo tempo que mantêm etapas que combinam variações aleatórias. Outro ponto interessante deste tipo de algoritmo é o armazenamento de soluções candidatas através da população. Desta maneira, a cada iteração (ou evolução da população), as soluções trocam informações entre si, encorajando a reprodução das "bem avaliadas" enquanto que "mal avaliadas" vão sendo eliminadas.

No SGA após iniciar a população aleatoriamente cada indivíduo é avaliado sendo que alguns serão selecionados. Neste passo temos a escolha dos pais que irão gerar os indivíduos da próxima população. Este operador genético – seleção – pode utilizar diversas técnicas. Entre elas temos o elitismo [5] que seleciona apenas os indivíduos melhor avaliados enquanto descarta os piores. Há também a seleção "por roleta" [65] onde os indivíduos são selecionados baseado na distribuição de probabilidade de sua aptidão.

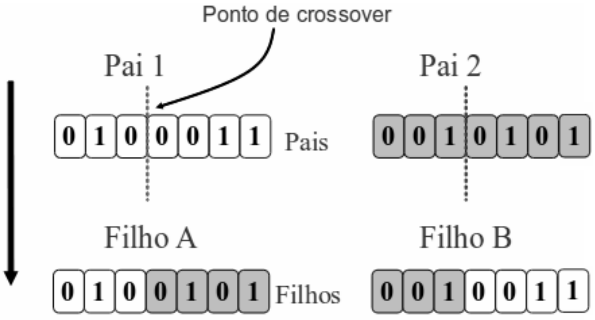


Figura 2.6: Recombinação de um ponto

Com os pais selecionados, o operador genético "recombinação"(*crossover* em inglês) é aplicado sobre eles representando o conceito de troca de informações entre soluções. Neste passo os pais selecionados são agrupados em pares onde cada par irá gerar dois filhos. Uma probabilidade p_c determina se a recombinação irá ocorrer. Portanto um número aleatório entre zero e um é gerado e caso este seja menor que a probabilidade de recombinação (exemplo: $p_c = 0{,}7$) a operação ocorre. Caso não ocorra os filhos serão idênticos aos pais.

A troca de informações dos pais é feita através de um (recombinação simples - exemplo figura 2.6) ou mais subconjuntos de bits. A posição e tamanho do subconjunto de bits podem ser escolhidos aleatoriamente no momento da operação ou pré-determinados na codificação do algoritmo.

Finalmente temos o operador mutação que é aplicado com baixa probabilidade p_m nos filhos produzidos pela recombinação (visto na figura 2.7). Caso este indivíduo seja escolhido para sofrer mutação, um *bit* de sua codificação é escolhido aleatoriamente e modificado de 0 para 1 ou de 1 para 0. A mutação tem papel importante na execução da busca por assegurar que qualquer ponto do espaço de busca tem possibilidade de ser atingido.

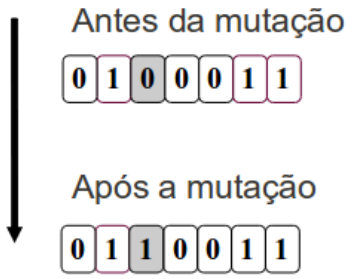


Figura 2.7: Mutação no terceiro *bit*

No algoritmo SGA clássico proposto por Holland [22] os indivíduos da população são codificados em valores binários de tamanho fixo. Michalewicz [65], ao utilizar este tipo de algoritmo em diversas aplicações reais, obteve resultados onde o desempenho não foi satisfatório. Seu trabalho indica que a representação binária fixa tem desempenho pobre nos problemas com alta dimensionalidade e onde a precisão é importante. O mesmo se aplica a problemas de otimização numérica com parâmetros reais. Por este motivo o RCGA (RCGA - *Real Coded Genetic Algorithm*) foi implementado neste trabalho. Assim foi possível verificar a influência da codificação do indivíduo na convergência do algoritmo nos problemas abordados.

2.2.2 GEO

O algoritmo de Otimização Extrema Generalizada (GEO – *Generalized Extremal Optimization*) é uma metaheurística recentemente proposta que também se baseia no conceito de seleção natural. Este algoritmo é uma generalização do método de Otimização Extrema [54]. Ambos os modelos tiveram seus processos internos inspirados na pesquisa de Bak-Sneppen [47] que propõe a presença da Criticalidade Auto Organizada (SOC – *Self-Organized Criticality*) em ecossistemas.

A teoria SOC propõe que sistemas grandes e complexos tendem a evoluir para um estado crítico na qual uma mudança em um único elemento produz perturbações que atingirão qualquer número de elementos do sistema. Uma lei de potência descreve a

probabilidade de uma perturbação de tamanho s ocorrer é:

$$P(s) \approx s^{-\tau} \tag{2.2}$$

onde τ é um número positivo. Observando esta função percebe-se que pequenas perturbações ocorrem com maior probabilidade mas perturbações que podem afetar o sistema inteiro podem acontecer com baixa probabilidade.

O modelo de Bak-Sneppen apresenta as espécies de um ecossistema alinhadas lado a lado de forma circular mantendo uma relação de vizinhança uma com as outras. A figura 2.8 representa um ecossistema de n espécies sendo e_1 e e_n espécies vizinhas.

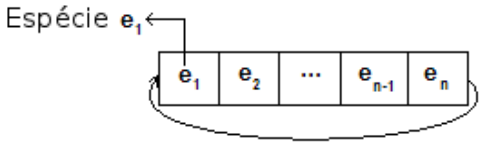


Figura 2.8: Espécies no modelo de Bak-Sneppen

Cada espécie recebe um valor aleatório entre [0,1] representando sua aptidão ao ecossistema. Valores baixos indicam espécies menos adaptadas. O sistema evolui forçando a mutação da espécie menos adaptada. Isso se reflete nas espécies vizinhas que também sofrem mutação pois agora encontram um novo competidor local. Este ciclo se repete de maneira que a aptidão média do ecossistema aumenta gradualmente. Isto acontece até que, em um certo momento, todas as espécies possuirão aptidão acima de um valor crítico. Nesta circunstância, como tanto espécies boas quanto ruins são forçadas a mudar, algumas acabam por ficar abaixo do nível critico. Esta perturbação no equilíbrio do ecossistema é chamada de avalanche e segue a regra de potência descrita na equação 2.2. Este modelo possibilita a construção de uma heurística de otimização que pode encontrar soluções rapidamente mudando sistematicamente as espécies menos adaptadas e escapando de mínimos locais através das avalanches.

Com a intenção de criar uma metaheurística que pudesse ser aplicada em problemas difíceis de otimização combinatória, Boettcher e Percus [54] propuseram o algoritmo EO (EO - *Extremal Optimization*). Apesar de obter um bom desempenho em alguns problemas seus próprios autores reconheceram que este método apresenta características que o impedem de ser utilizado em outras grandes classes de problemas. A maior delas é o fato de que fica a cargo do implementador a definição do índice de adaptação de cada variável do projeto.

Com a intenção de sanar esses problemas o algoritmo GEO foi criado para permitir sua utilização em uma grande classe de problemas com qualquer tipo de variável, contínua,

discreta ou uma mistura delas. No algoritmo GEO uma população é representada por uma sequência (*string*) de bits onde cada bit é uma espécie. Isto difere do algoritmo genético (SGA) onde cada espécie é composta por uma sequência de bits representando uma possível solução do problema. Já no EO cada variável do projeto é uma espécie. Esta diferenciação afeta a maneira como a avaliação da função objetivo é trabalhada pois este valor é associado à espécie da população. A figura 2.9 apresenta como duas variáveis, x e y, são representadas como espécies no GEO, EO e SGA.

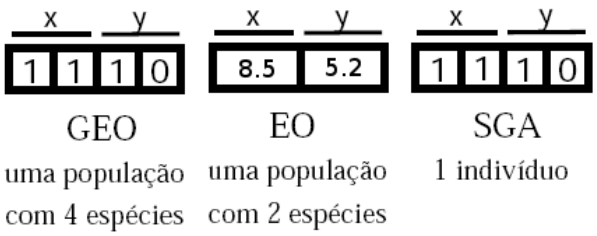


Figura 2.9: Espécies representadas em GEO, EO e SGA

Uma descrição dos passos do algoritmo GEO é apresentada resumidamente abaixo na figura 2.10 e a partir do texto adaptado do trabalho de Sousa [36]:

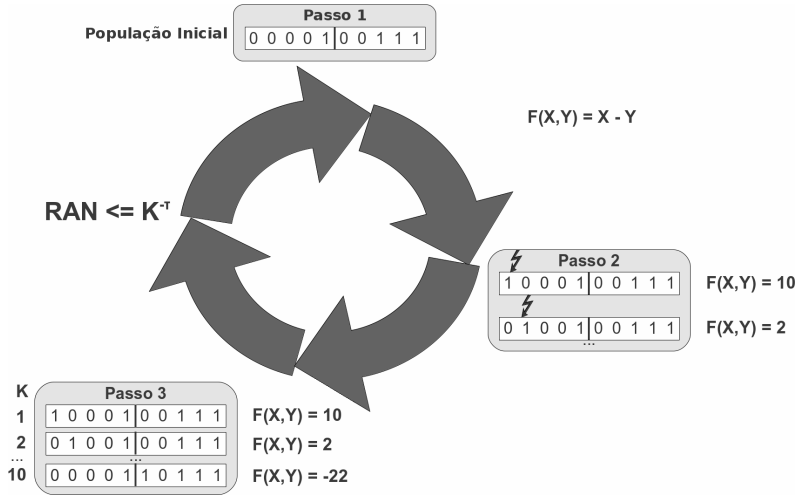


Figura 2.10: Funcionamento do GEO

1. Inicialize aleatoriamente uma *string* de bits de tamanho L que codifica N variáveis do problema. Para o exemplo da figura 2.10 têm-se duas variáveis (N=2) onde a precisão é de um inteiro indo de 0 à 31. Portanto cada variável pode ser codificada por 5 bits (L=10).

2. Mude o bit i dentro da população e calcule o valor da função objetivo (para a figura 2.10 a função objetivo apresentada serve apenas como exemplo) para esta nova população. Atribua ao *bit* um índice de adaptabilidade de acordo com o ganho ou perda adquirida com esta mutação em relação ao melhor valor da função objetivo encontrada até o momento. Retorne o *bit* ao valor original.

3. Repita o passo acima para todos os *bits* da população como mostrado no passo 2 da figura 2.10 (i vai de 1 à 10).

4. Ordene os *bits* de acordo com seus índices de adaptabilidade, de K=1 para o *bit* mais adaptado até K=L para o menos adaptado. Ordene os *bits* aleatoriamente com distribuição uniforme caso dois ou mais tenham um índice de adaptabilidade igual (passo 3 da figura 2.10).

5. Escolha com probabilidade uniforme um *bit* candidato para sofrer mutação (escolha um K aleatoriamente). Gere um número aleatório RAN com probabilidade uniforme entre [0,1]. Calcule $P_k = K^{-\tau}$, sendo K a posição e τ um valor ajustável. Caso P_k seja igual ou maior que RAN mute o *bit* - ou seja - substitua a população corrente pela população temporária K. Senão repita o processo até que um *bit* seja modificado.

6. Repita os itens de 2 a 4 até que um critério de parada seja satisfeito.

7. Retorne a melhor solução (configuração de *bits*) encontrada.

Antes de implementar o algoritmo GEO deve-se definir o valor da variável τ. Com apenas esta variável ajustável o algoritmo pode se tornar tanto uma busca aleatória como uma busca local. Quando $\tau \to 0$ qualquer candidato escolhido sofrerá mutação levando a uma busca aleatória. Quando $\tau \to \infty$ somente o primeiro bit sofrerá mutação levando a uma busca local. Trabalhos anteriores com o algoritmo GEO mostraram que o valor de τ que traz os melhores resultados geralmente está entre [0,10] [59, 38]. É interessante notar que com apenas uma variável de ajuste o GEO se destaca em relação a diversos outros algoritmos. No caso de um SGA clássico temos que ajustar o tamanho da população, a probabilidade de recombinação e a probabilidade de mutação. Outros parâmetros como domínio de entrada, critério de parada e precisão são comuns a todos os algoritmos e não interferem diretamente no desempenho do algoritmo durante sua execução.

O algoritmo descrito anteriormente também é chamado de GEO canônico. Existe uma variação chamada de GEOvar onde os *bits* são ordenados separadamente para cada codificação de cada uma das variáveis do projeto. Na figura 2.11 vemos uma comparação das principais características dos algoritmos GEOcan e GEOvar retirados de [37].

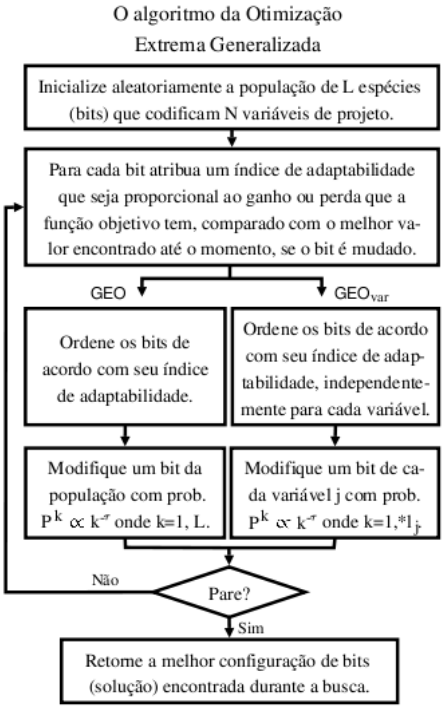


Figura 2.11: Comparação entre GEOcan e GEOvar

2.2.3 GEOreal

Uma nova versão do GEO utilizando variáveis reais ao invés de *string* binária é chamada GEOreal. Este novo modelo tenta corrigir uma limitação do algoritmo GEO que ao utilizar a codificação binária obriga o implementador a definir a precisão (número de *bits*) das variáveis. A mutação de um *bit* (de 0 para 1 ou vice-versa) causa um salto em um conjunto de valores intermediários que podem acabar não sendo testados. Além disso, o algoritmo pode não encontrar o mínimo local por este não pertencer ao conjunto de soluções possíveis de serem testadas [35].

O algoritmo GEOreal segue o mesmo princípio básico do GEO. A principal mudança é que agora o valor da variável de projeto é atualizada criando-se uma perturbação através de um número aleatório com distribuição gaussiana. O GEOreal segue os seguintes passos [35]:

1. Inicialize aleatoriamente uma população de N espécies onde cada variável de projeto é representada por uma espécie.

2. Uma por vez, altere cada variável do projeto segundo a equação 2.3. x'_i é o novo valor da i-éssima variável e N(0,s) é um número aleatório com distribuição gaussiana de média zero e desvio padrão s. Calcule V_i (valor da avaliação da população

temporária com a variável de projeto i modificada) e gere o par ordenado (população temporária,V_i). Armazene este par e retorne o valor de x_i para antes da perturbação.

$$x'_i = x_i + N(0, \sigma)x_i \tag{2.3}$$

3. Ordene as populações temporárias, de 1 a N, de acordo com seu V_i atribuindo um rank k para cada população. A menos adaptada recebe k=1 e a melhor adaptada recebe k=N.

4. Escolha com probabilidade uniforme uma população temporária. Gere um número aleatório RAN com probabilidade uniforme entre [0,1]. Calcule $P_i = k^{-\tau}$. Caso P_i seja igual ou maior que RAN substitua a população corrente (da iteração) pela população temporária. Caso contrário repita este passo até que uma população temporária seja escolhida.

5. Repita os passos 2 a 4 até que algum critério de parada seja atingido.

6. Retorne a melhor função objetivo encontrada e o conjunto de melhores variáveis.

No trabalho de Lopes [35] o algoritmo GEOreal, quando comparado com o GEOcan e GEOvar nos problemas abordados, obteve bons resultados. Esta nova implementação (GEOreal) foi capaz de alcançar o mínimo global e apresentar um melhor desempenho que o GEOcan.

Outra implementação possível a partir do algoritmo GEO é o GEOint. Segue-se a mesma ideia da implementação GEOreal mas utilizando a codificação inteira. A perturbação também é aplicada utilizando-se um número aleatório com distribuição gaussiana.

Capítulo 3

Trabalhos Existentes

Como comentado na seção 2, a qualidade de um *software* está diretamente ligada ao ciclo de Verificação e Validação. Testes fazem parte deste processo e a maneira como estes são aplicados consequentemente determinam um bom produto final. A seleção e execução manual de testes tem se demonstrado ser caro, trabalhoso e imperfeito. Pesquisas recentes [59, 39, 44] demonstraram que a geração automática de dados de teste utilizando metaheurísticas são mais confiáveis do que processos manuais ou testes utilizando valores aleatórios. Um exemplo é o estudo de McMinn [49] que mostra diversos tipos de metaheurísticas atualmente utilizadas tanto em testes estruturais quanto em funcionais.

Apesar disso o grande número de trabalhos na área de geração automática de dados para testes aponta que não existe uma metodologia única para lidar com este problema [40, 49]. A maneira como o teste de *software* será aplicado e a metaheurística escolhida são variáveis que devem ser escolhidas dependendo da classe de problemas que serão atacados. No contexto de otimização, o desempenho do algoritmo está diretamente relacionado à escolha da função objetivo e do seu domínio de busca.

Wolpert et al. [24] mostraram em seu trabalho que se um algoritmo tem em média um bom desempenho em uma classe de problemas, para uma outra classe, em média, seu desempenho é pior. Portanto, mesmo técnicas já atualmente dedicadas a determinados problemas não são extensíveis a novos tipos de problemas sendo necessário uma nova solução para estes. Segundo esse contexto, a utilização de algoritmos evolutivos para a geração automática de dados de testes é uma área passível de pesquisa.

A próxima seção apresenta os diversos trabalhos que abordam a geração de dados de teste utilizando métodos de busca global. Também são apresentados trabalhos que abordam o uso das funções objetivo. Na seção seguinte é feito um detalhamento do funcionamento das funções objetivo utilizadas nesta pesquisa.

3.1 Trabalhos

Muitas pesquisas recentes fazem o uso de métodos globais de busca [14, 50, 3, 63, 58, 17, 48, 31, 6] para abordar o problema de geração de dados de teste. Este campo de pesquisa também tem sido chamado de SBT (SBT – *Search Based Testing*). De forma geral um algoritmo é utilizado para cobrir um determinado critério de teste, mensurado através de um tipo de cobertura do problema. Alguns desses critérios incluem a cobertura de caminhos [58, 48, 3, 63] enquanto outros focam na cobertura de instruções e condicionais [50, 17, 31, 14]. Em ambos os casos uma função objetivo [48, 13, 3, 63] é utilizada para mapear as soluções do método de busca ao critério de cobertura, aferindo desta maneira uma aptidão às soluções.

As próximas subseções apresentam primeiramente os trabalhos que focam no tipo de algoritmo utilizado para gerar dados de teste. Na subseção seguinte discute-se os tipos de função objetivo existentes.

3.1.1 Métodos de Busca Global

Muitas são as opções dentre os métodos de busca global utilizados para a geração automática de dados de teste. Alba et al. [17] aplicam estratégias evolutivas (EE) nesta tarefa enquanto que Korel [6] utiliza o método de gradientes. Neste último, o objetivo é cobrir caminhos selecionados, mesmo critério abordado nesta pesquisa. Korel utiliza a análise dinâmica de fluxo de dados do SUT executado para identificar o predicado onde houve o desvio do caminho desejado. Desta maneira esta informação era utilizada para que o algoritmo encontrasse um dado de entrada que pudesse corrigir este desvio na direção do objetivo. Windish et al. [4] utilizam um método de busca global conhecido como enxame de partículas (PSO - *Particle Swarm Optimization*) para abordar a geração de dados de teste utilizando a técnica caixa branca. O critério de cobertura utilizado é o de predicados e o estudo mostrou bons resultados para o algoritmo PSO.

Apesar disto a maioria dos trabalhos publicados [58, 48, 31, 14, 50, 3, 43, 63] apontam a utilização de algoritmos genéticos como gerador de dados de entrada para casos de testes. A próxima seção apresenta alguns desses trabalhos.

Algoritmos Genéticos

Nos trabalhos que abordam o uso do algoritmo genético para a geração de dados de teste, este funciona como um otimizador utilizando as variáveis de entrada do caso de teste como espécies de uma população. A partir daí os operadores genéticos são aplicados enquanto uma função objetivo conduz à cobertura desejada do teste. Por este motivo é importante definir qual técnica de teste e qual critério de cobertura será utilizado. Há

trabalhos que focam na utilização de metaheurísticas em problemas de teste funcional [1, 52] mas que não serão abordados por fugirem do contexto desta pesquisa. Há também aqueles [17, 31, 14, 50, 43] que abordam o teste estrutural e que utilizam critérios como o de instruções, caminhos e condicionais. A pesquisa de Michael et al. [14] é um exemplo deste último critério. Nele é sugerido o uso de uma função de minimização baseada em gradiente como função objetivo. A cobertura dos testes é feita em todos os condicionais forçando que toda decisão falsa e verdadeira seja verificada.

A pesquisa de Pargas et al. [50] também utiliza a técnica de teste estrutural e propõe a utilização de um AG para a cobertura de instruções e predicados. Este trabalho foi desenvolvido especificamente para problemas muito complexos. Por este motivo, para que a otimização obtenha sucesso, uma combinação dos benefícios de algoritmos aleatórios e algoritmos genéticos é utilizada. Além disso, a possibilidade de aproveitar a distribuição em multiprocessadores permite a diminuição considerável do tempo de execução. A avaliação da função objetivo baseia-se principalmente no modelo de grafos de dependência de con-trole (GDC). Isto permite que seja possível identificar os predicados para o caminho que se deseja cobrir. Ao se executar o algoritmo com dados de entrada, os predicados percorridos são armazenados e posteriormente comparados com os predicados do caminho desejado. A função objetivo atribui um maior valor para ca-minhos que contêm mais predicados em comum. Informações sobre os comandos de desvios também são utilizados para permitir que valores mais precisos sejam obtidos.

Apesar dos diversos trabalhos utilizando GA existe uma metaheurística recente conhecida como GEO que tem demonstrado ser simples por possuir apenas uma variável de ajuste. Este algoritmo foi escolhido para ser utilizado nesta pesquisa e alguns dos trabalhos que utilizam o GEO são mencionados na próxima subseção.

GEO

Apesar de recente o algoritmo GEO já foi estudado em alguns trabalhos para abordar a geração automática de dados de teste. Abreu [58] utiliza este algoritmo ao invés de um GA na geração de dados de teste estrutural. O GEO foi combinado com a função objetivo *Similarity* [13] que avalia a semelhança entre um caminho executado e um alvo. Os resultados foram comparados com um SGA e geração aleatória concluindo que o GEO exigiu menos esforço computacional (uso de recursos como memória, processador, etc) para gerar os mesmo dados.

Apesar da contribuição da pesquisa de Abreu [58, 60, 59] existem novas implementações do GEO que não foram avaliadas - como o GEOreal - e apenas uma função objetivo foi pesquisada. Por este motivo a pesquisa desenvolvida neste trabalho de mestrado teve como propósito implementar mais uma função objetivo para assim observar a influência desta na geração de dados de testes.

Outro trabalho que utiliza o GEO na geração de dados de teste é o de Yano [64]. Aqui a técnica de testes utilizada é a funcional pois o proposto M-GEO, GEO multiobjetivo, é utilizado para gerar sequências de eventos que conduzam uma máquina de estados finitos estendida (EFSM - *Extended Finite State Machine*) à um determinado estado desejado. Este EFSM é responsável por representar os comportamentos possíveis de um determinado sistema real e o estado que se deseja alcançar é gerado pelo M-GEO que é baseado no conceito de ótimo de Pareto[1].

Neste trabalho não é abordado o conceito multiobjetivo e apenas uma FO é utilizada pelo algoritmo GEO. Por este motivo a próxima seção apresenta trabalhos que focam na pesquisa das funções objetivos e suas características.

3.1.2 Função Objetivo

A função objetivo é responsável por guiar a metaheurística na geração de dados de teste dinâmica. Por este motivo alguns trabalhos dedicam-se a avaliar diversas FOs. Wegener et al. [63] comparam a utilização de duas funções para um mesmo problema. Neste caso, o problema estudado foi um sistema autônomo de estacionamento. A simulação é feita a partir de uma vaga de estacionamento e de um carro que se aproxima desta vaga. O algoritmo genético deve gerar cinco dados de teste para estacionar o carro corretamente na vaga sem colidir. A primeira função objetivo utiliza a distância entre o carro e região de colisão para avaliar um valor. Já a segunda função objetivo utiliza a área entre o carro e a região de colisão. Em ambos os casos, se houver alguma colisão o valor da função objetivo se torna negativa. Caso o veiculo consiga apenas tocar na guia o valor da função objetivo é zero. Qualquer outra manobra diferente destas atinge um valor positivo e o objetivo é minimizar este valor. O trabalho concluiu que a escolha da função objetivo afeta diretamente a maneira como o algoritmo genético trabalha. Neste caso a função objetivo baseada na área entre o veículo e a região de colisão mostrou ser mais efetiva na transição de valores negativos para positivos enquanto que a baseada na distância obtinha valores constantes que não traziam benefício para a evolução da população do algoritmo genético.

Outra pesquisa que foca na função objetivo é o trabalho de Lakhotia et al. [31]. Duas implementações dos algoritmos genéticos são utilizadas em conjunto com uma função multi-objetivo. A primeira parcela da função utiliza o conceito de distância e nível de aproximação. O modelo é o de "proximidade" de Wegener [30] que reflete a distância de dois nós a partir de um nó condicional de um CFG procurando pelo caminho mais "rápido". Já a segunda parcela da função multi-objetivo tenta minimizar a quantidade

[1]Ótimo de Pareto é um conjunto que contém apenas soluções não dominantes. Dominação é definido como um indivíduo X domina Y se, e somente se, X é melhor que Y em pelos menos um objetivo, e não é pior que todos os outros objetivos

de memória utilizada durante a execução do algoritmo.

No trabalho de Bueno e Jino [48] uma função objetivo é utilizada para guiar a busca de um GA. Esta FO leva em consideração dois fatores: quantos nós do caminho percorrido são comuns aos do caminho alvo e em qual condicional houve o desvio do caminho. Cada indivíduo é avaliado então através do seguinte cálculo: um número inteiro que indica o número de nós comuns entre os caminhos (percorrido e alvo) partindo da primeira aresta até onde ocorreu o desvio reduzido-se de uma penalidade. Esta penalidade é calculada analisando-se os dados de teste. Quanto maior for a distância do dado de teste em satisfazer o condicional onde ocorreu o desvio, maior será a penalidade. Desta maneira a função objetivo de Bueno e Jino [48] visa maximizar o número de arestas comuns entre os caminhos e minimizar a penalidade dos dados de testes no nó condicional onde houve o desvio.

Método	**Método de otimização principal**	**Função objetivo**	**Critério de teste**
[50]	AG	Avaliação da semelhança entre predicados cobertos	Instruções e predicados
[48]	AG	Semelhança de caminhos e penalidade	Caminhos
[63]	AG	Área/distância entre carro e área de colisão	Caminhos
[58]	GEO	Semelhança de caminhos	Caminhos
[3]	AG	Probabilidade inversa do caminho	Caminhos
[43]	AG	Semelhança de caminhos e penalidade associada aos predicados	Caminhos
[31]	AG	Multiobjetivo - distância e nível de aproximação	Predicados
[17]	EE	Avaliação de funções associados aos predicados	Condicionais
[4]	PSO	Avaliação de funções associados aos predicados	Predicados
[14]	AG	Avaliação de funções associada aos predicados	Condições e decisões

Tabela 3.1: Trabalhos sobre testes evolutivos

Mais recentemente Watkins [3] propôs em sua pesquisa uma função objetivo chamada de IPP (IPP - *Inverse Path Probability*) e a comparou com outras FOs conhecidas, entre elas a de Mansour e Salame [43], *Similarity* [13] e Bueno e Jino [48]. A IPP apenas

conta quantas vezes cada caminho foi percorrido e o valor da aptidão de cada indivíduo é o inverso deste número. Sendo assim, quanto mais um caminho for repetido mais o indivíduo que o repetiu será penalizado. Desta maneira esta FO incentiva a diversidade de caminhos pois os caminhos menos percorridos são os dos indivíduos mais aptos da população. Foi a partir desta pesquisa que a função objetivo de Bueno e Jino foi escolhida para ser implementada e comparada com a *Similarity*.

A tabela 3.1 mostra resumidamente todos os trabalhos descritos nesta seção comparando o principal método de otimização utilizado, a função objetivo e o tipo de critério de teste.

3.2 Detalhamento das FOs

Esta pesquisa utilizou duas conhecidas FOs: a *Similarity* [13] e a proposta por Bueno e Jino [48]. Esta seção tem como objetivo detalhar seu funcionamento.

3.2.1 Função Objetivo *Similarity*

Proposta por Lin e Yeh [13] também é conhecida como NEHD (NEHD – *Normalized Extended Hamming Distance*). Esta FO foi utilizada no estudo de Abreu [58] (de onde este trabalho partiu) e foi novamente implementada para que os resultados obtidos pudessem ser comparados.

Dois caminhos, corrente e alvo, são avaliados e um valor real é retornado indicando a similaridade entre estes. Para calcular este valor a FO executa diversas operações matemáticas de conjunto, quanto maior for a similaridade, maior o valor retornado.

Esta função objetivo possui vantagens e limitações conhecidas. Entre suas vantagens esta o fato da simplicidade de implementar a FO já que suas operações internas apenas tratam cálculos e operações de conjuntos. Outra vantagem é que o nível de abstração esconde a complexidade interna do SUT, ou seja, não é relevante a complexidade de um predicado onde ocorreu um desvio do caminho alvo, importa apenas cobrir este predicado. Por fim, esta FO tem como ponto forte o fato de receber como entrada apenas os dois caminhos (alvo e corrente) o que simplifica muito a quantidade de informações que devem ser obtidas da instrumentação do SUT. Como limitação é conhecido [3, 58] que a função objetivo não trata caminhos com *loops*. Isso foi observado nesta pesquisa ao verificar que dois caminhos com diferente número de iterações em um *loop* foram avaliados com mesmo valor.

Cálculo da *Similarity*

Abaixo são descritos alguns conceitos importantes para compreender o cálculo da *Similarity*. Cada um dos itens apresenta um conceito interno do cálculo da função objetivo. O último item (similaridade total) apresenta o valor final da similaridade computada pela função objetivo. Sendo assim, para melhor compreender estes itens, dois exemplos de caminhos serão utilizados (cada letra refere-se a uma aresta de um CFG) :

caminho α - abc
caminho β - abccc

1. Conjunto de arestas é um conjunto de arestas distintas de um caminho. O número de elementos varia de acordo com a ordem do conjunto.

2. Ordem do conjunto é o número de arestas combinadas de um conjunto (tamanho do conjunto). Veja exemplo nas tabelas 3.2 e 3.3.

Notação	**Conjuntos de arestas**	**Ordem do Conjunto**
$C^1{}_\alpha$	a,b,c	3
$C^2{}_\alpha$	ab,bc	2
$C^3{}_\alpha$	abc	1

Tabela 3.2: Conjunto de arestas para o caminho α

Notação	**Conjuntos de arestas**	**Ordem do Conjunto**
$C^1{}_\beta$	a,b,c,**c**,**c**	3
$C^2{}_\beta$	ab,bc,cc,**cc**	3
$C^3{}_\beta$	abc,bcc,ccc	3
$C^4{}_\beta$	abcc,bccc	2
$C^5{}_\beta$	abccc	1

Tabela 3.3: Conjunto de arestas para o caminho β

Os conjuntos de arestas marcados em **negrito** são eliminados pois representam conjuntos já existentes. Portanto em $C^1{}_\beta$ ficamos apenas com os conjuntos a,b,c. Isso acaba por eliminar a presença de *loops* no caminho.

3. Ordem máxima de um conjunto é o número de arestas distintas em um caminho. Sua representação é $O^{MAX}{}_x$, onde x é o caminho. Para os exemplos teremos: $O^{MAX}{}_\alpha = 3$ e $O^{MAX}{}_\beta = 3$ (pois eliminando-se as repetições teremos apenas a,b,c).

4. A distância simétrica entre conjuntos de uma mesma ordem k é representado por $D^{k}{}_{x,y}$ onde x e y são os caminhos. O cálculo é feito unindo-se todos os elementos destes conjuntos (eliminando elementos repetidos) e excluindo-se os elementos comuns. Para o nosso exemplo:
$$D^{2}{}_{\alpha,\beta} = (C^{2}{}_{\alpha} \cup C^{2}{}_{\beta}) \text{ - } (C^{2}{}_{\alpha} \cap C^{2}{}_{\beta})$$
$$D^{2}{}_{\alpha,\beta} = \cup^{2}{}_{\alpha,\beta} \text{ - } \cap^{2}{}_{\alpha,\beta}$$
$$D^{2}{}_{\alpha,\beta} = \{ab,bc,cc\} \text{ - } \{ab,bc\}$$
$$D^{2}{}_{\alpha,\beta} = \{cc\}$$

5. A distância normalizada entre conjuntos é um valor real entre [0,1]. O seu cálculo é dado por:
$$DN^{k}{}_{x,y} = \frac{|D^{k}{}_{x,y}|}{|\cup^{k}{}_{x,y}|}$$
$$DN^{k}{}_{x,y} = \frac{|\cup^{k}{}_{x,y} - \cap^{k}{}_{x,y}|}{|\cup^{k}{}_{x,y}|}$$
$$DN^{k}{}_{x,y} = 1 - \frac{|\cap^{k}{}_{x,y}|}{|\cup^{k}{}_{x,y}|}$$
Podemos perceber que se $|\cap^{k}{}_{x,y}| = |\cup^{k}{}_{x,y}|$ então $DN^{k}{}_{x,y} = 0$. Da mesma maneira se $|\cap^{k}{}_{x,y}|$ for vazio, $DN^{k}{}_{x,y} = 1$.
Para o nosso exemplo:
$$DN^{2}{}_{\alpha,\beta} = \frac{1}{3} = 0,3334$$

6. Similaridade entre conjuntos de arestas indica a proximidade entre conjuntos de uma determinada ordem através de um número real entre [0,1]. A equação é dada por:
$$S^{k}{}_{x,y} = \frac{|\cap^{k}{}_{x,y}|}{|\cup^{k}{}_{x,y}|}$$
$$S^{k}{}_{x,y} = 1 \text{ - } DN^{k}{}_{x,y}$$
Caso não haja elemento comum entre conjuntos de uma ordem k o valor de $S^{k}{}_{x,y}$ vale zero. Por outro lado se todos os elementos dos conjuntos forem idênticos então $S^{k}{}_{x,y} = 1$. Para o nosso exemplo:
$$S^{2}{}_{\alpha,\beta} = 1 - \frac{1}{3} = \frac{2}{3} = 0,6667$$

7. Fator de peso é um valor multiplicado à similaridade entre conjuntos. Este fator é proporcional à ordem da similaridade entre conjuntos. Seu cálculo é dado por:
$$P^{k}{}_{x} = |C^{k\text{-}1}{}_{x}| * P^{k\text{-}1}{}_{x}$$
Onde $P^{1}{}_{x} = 1$, k = 2,...,$O^{MAX}{}_{x}$. A tabela 3.4 aplica o fator de peso para os exemplos.

8. A similaridade total entre caminhos é a soma total de todas as similaridades de todas as ordens entre os dois caminhos, onde similaridade é multiplicada pelo fator de peso da ordem.

k	Cálculo	P^k
1	1, por definição	1
2	$\|C^1{}_\alpha\| * P^1{}_\alpha = 3 * 1$	3
3	$\|C^2{}_\alpha\| * P^2{}_\alpha = 3 * 3$	9

Tabela 3.4: Fator peso para o caminho α

$$SIM_{x,y} = S^1 * P^1{}_x + S^2 * P^2{}_x \ldots + S^{OMAX} * P^{OMAX}{}_x$$

Para melhor entender os cálculos abaixo são mostrados dois exemplos:

caminho α - abcel

caminho β - abfhil

$SIM_{\alpha,\beta}$ = 1*0.6 + 8*0.4 + 56*0.33 + 336*0.25 + 1680*0.14

$SIM_{\alpha,\beta}$ = **341.48**

caminho α - abcfcfcfiijm

caminho β - abcfcfhijkl

$SIM_{\alpha,\beta}$ = 1*0.6 + 9*0.4 + 72*0.2 + 504*0.11

$SIM_{\alpha,\beta}$ = **74.04**

3.2.2 Função Objetivo Bueno e Jino

Foi proposta por Bueno e Jino [48] e combina informações do fluxo de controle e de dados a fim de maximizar o número de arestas entre dois caminhos, o percorrido e o alvo, e minimizar uma penalidade aplicada no predicado onde ocorreu o desvio do caminho alvo.

De forma geral o cálculo pode ser dado pela seguinte equação:

$$F_t = NC - \left(\frac{EP}{MEP}\right) \tag{3.1}$$

1. F_t é o valor da avaliação da FO. Esta função deve ser maximizada e o valor máximo é quando ambos os caminhos (percorrido e alvo) forem iguais.

2. NC é o número de arestas em comum entre o caminho percorrido e o caminho alvo. Esta contagem é feita a partir da primeira aresta.

3. EP é o *módulo* do erro ocorrido no predicado onde o caminho percorrido se difere do alvo. Este valor é obtido a partir da análise de fluxo de dados dinâmica, ou seja, das informações obtidas através da instrumentação inserida no código executado. Esta análise informa, por exemplo, quais instruções foram executadas, etc.

4. MEP também é obtido a partir da análise de fluxo de dados dinâmica e representa o maior erro obtido por todas as execuções que desviaram do caminho alvo no mesmo predicado.

Uma análise mais detalhada da equação mostra que a segunda parcela (EP / MEP) sempre será um valor entre [0,1]. Como esta segunda parcela é uma penalidade aplicada sobre a primeira (NC), esta será máxima quando EP for igual a MEP, subtraindo o valor de 1 de NC. Quando os dois caminhos forem iguais (alvo e percorrido) esta segunda parcela será zero.

Outro fato relevante e que deve ser detalhado é o fator MEP. Para se encontrar este valor deve-se utilizar a análise de fluxo de dados dinâmica de várias execuções do SUT. Nota-se portanto um grande diferencial desta FO em relação à *Similarity*. Enquanto esta última precisa apenas dos dois caminhos (alvo e percorrido) para o cálculo, a FO proposta por Bueno e Jino depende de uma análise de diversas execuções. Aplicando-se isto à realidade dos algoritmos evolutivos temos a seguinte compreensão: NC e EP podem ser calculados a partir das entradas do caminho alvo e da execução (caminho percorrido) de um indivíduo da população. Já MEP só pode ser calculado com a execução (caminhos percorridos) por todos os indivíduos da população. Portanto, para calcular o valor da avaliação de cada indivíduo da população precisamos antes da execução e da análise de fluxo dinâmica de todos os indivíduos da população.

Para computar o valor de EP deve-se incluir uma instrumentação específica em cada predicado do SUT. Predicados simples são aqueles que possuem apenas uma expressão relacional (equação ou inequação) do tipo EA1 *oper* EA2, onde EA1 e EA2 são expressões aritméticas e $oper \in \{<, \leq, >, \geq, =, \neq\}$. Predicados compostos são combinações de predicados simples utilizando os operadores OR e AND [7]. Todo predicado simples pode ser transformado na forma F rel 0, onde $rel \in \{<, \leq, =, \neq\}$. Esta função é denominada *função de predicado* (EP). Como exemplo temos:

$$a < b$$

Será transformado em:

$$a - b < 0$$

Percebe-se que a função é positiva quando o predicado é falso e negativa quando verdadeira. Desta maneira a função reflete o estado das variáveis de entrada. Desta maneira podemos minimizar a função de predicado manipulando as variáveis de entrada para torná-lo verdadeiro.

A tabela 3.5 apresenta as diversas funções de predicado para cada tipo de predicado. Em especial temos o caso onde EA1 $\neq$ EA2. Nesta situação a função de predicado é satisfeita com F $\neq \frac{1}{k1}$ sendo k1 um valor entre 0 e 1. Isto reflete que quanto maior $|EA1–EA2|$ melhores são as soluções.

Predicado	Função Predicado
$EA_1 > EA_2$	$F = EA_2 - EA_1$
$EA_1 \geq EA_2$	$F = EA_2 - EA_1$
$EA_1 < EA_2$	$F = EA_1 - EA_2$
$EA_1 \leq EA_2$	$F = EA_1 - EA_2$
$EA_1 = EA_2$	$F = \lvert EA_1 - EA_2 \rvert$
$EA_1 \neq EA_2$	$F = \frac{1}{\lvert EA_1 - EA_2 \rvert + k_1}$

Tabela 3.5: Função de predicado para cada tipo de predicado

Para predicados compostos utiliza-se a seguinte regra:

$$\mathrm{F}(\mathrm{cond}_1 \text{ AND } \mathrm{cond}_2) = \mathrm{F}(\mathrm{cond}_1) + \mathrm{F}(\mathrm{cond}_2)$$
$$\mathrm{F}(\mathrm{cond}_1 \text{ OR } \mathrm{cond}_2) = \mathrm{MIN}(\mathrm{F}(\mathrm{cond}_1),\mathrm{F}(\mathrm{cond}_2))$$

Cálculo da FO Bueno e Jino

Para demonstrar o funcionamento da função objetivo Bueno e Jino será utilizado o SUT "valor do meio". O CFG deste programa pode ser visto na figura 3.1.

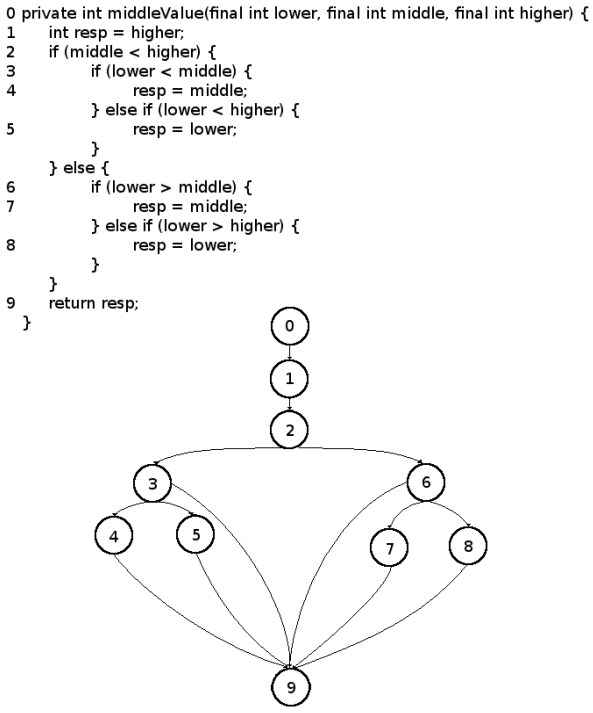


Figura 3.1: CFG do SUT valor do meio

Este programa possui quatro caminhos independentes que podem ser vistos na tabela 3.6.

Caminho	**Sequência**
1	0,1,2,3,4,9
2	0,1,2,3,5,9
3	0,1,2,6,7,9
4	0,1,2,6,8,9

Tabela 3.6: Caminhos independentes para o SUT valor do meio

Vamos tomar como caminho alvo o número 2: 0,1,2,3,5,9. A partir disto suponhamos que houveram 10 execuções do SUT. A tabela 3.7 reflete estas execuções. A sequência de números sublinhada representa as arestas percorridas pela execução em comum com o caminho alvo.

Execução	**Caminho**	**Sequência**	**Valores de entrada**
1	1	0,1,2,3,4,9	1,5,8
2	3	0,1,2,6,7,9	10,5,4
3	1	0,1,2,3,4,9	2,9,10
4	1	0,1,2,3,4,9	1,2,5
5	2	0,1,2,3,5,9	5,2,6
6	4	0,1,2,6,8,9	3,5,2
7	4	0,1,2,6,8,9	2,8,2
8	4	0,1,2,6,8,9	6,7,1
9	4	0,1,2,6,8,9	2,3,1
10	3	0,1,2,6,7,9	4,3,2

Tabela 3.7: Execuções do SUT valor do meio

Em seguida são calculadas as parcelas da FO. Elas são apresentadas na tabela 3.8.

Execução	**NC**	**EP**	**MEP**	**EP/MEP**	$\mathbf{F_t}$
1	4	$\|1-5\| = 4$	7	0,57	3,43
2	3	$\|5-4\| = 1$	6	0,16	2,84
3	4	$\|2-9\| = 7$	7	1	3
4	4	$\|1-2\| = 1$	7	0,14	3,86
5	6	$\|0\| = 0$	0	0	**6**
6	3	$\|5-2\| = 3$	6	0,5	2,5
7	3	$\|8-2\| = 6$	6	1	2
8	3	$\|7-1\| = 6$	6	1	2
9	3	$\|3-1\| = 2$	6	0,33	2,67
10	3	$\|3-2\| = 1$	6	0,16	2,84

Tabela 3.8: Parcelas do cálculo da FO Bueno e Jino

Ambas as funções objetivo descritas nesta seção foram utilizadas no estudo empírico deste trabalho. A próxima seção apresenta como este estudo foi conduzido utilizando estas FOs através de diversos algoritmos.

Capítulo 4

Estudo Empírico

Dada a importância de se estudar novas metaheurísticas na geração de dados de teste este capítulo tem como objetivo avaliar em especial o algoritmo GEOreal combinado com as duas funções objetivo comentadas na seção 3. Este algoritmo possui a vantagem de ter sua população codificada no formato real o que, em outras pesquisas sobre algoritmos evolutivos [65, 35], mostrou possuir vantagens frente à codificação binária. O desempenho do GEOreal será avaliado aplicando-o sobre um conjunto de SUTs - detalhados na próxima seção - em conjunto com outros algoritmos conhecidos.

O uso de duas funções objetivo neste estudo empírico tem como objetivo avaliar o impacto da escolha da FO ao abordar a geração de dados de teste. Não é o mesmo objetivo do trabalho de Watkins [3] onde diversas FOs foram comparadas para se encontrar uma mais eficiente. Apesar disto, como já foi explicado na seção 3 o estudo de Watkins foi utilizado para se escolher a FO Bueno e Jino pelo seu bom desempenho.

A próxima seção inicia explicando quais e como os SUTs foram escolhidos para este estudo. A seção seguinte descreve a estratégia utilizada para executar os algoritmos sobre os SUTs. Na seção 4.3 apresenta-se como foi conduzido o ajuste dos parâmetros dos algoritmos antes deles serem executados sobres os SUTs. A seção 4.4 mostra os experimentos deste estudo empírico e seus resultados. Por fim a última seção contém uma discussão sobre estes resultados.

4.1 Programas Alvo

Para este estudo empírico os SUTs escolhidos para serem estudados são os mesmo utilizados por autores dos trabalhos [14, 50, 23, 43, 58] e são listados na tabela 4.1. O critério de cobertura utilizado é o de caminhos selecionados pelo usuário. Por este motivo para cada SUT um subconjunto de caminhos foi escolhido dada a restrição explicada na seção 2.1.1. Dado que o foco desta pesquisa não é a maneira como estes caminhos devem ser

selecionados eles foram escolhidos aleatoriamente.

#	SUT	NIV[a]	Tipo	Domínio	NTP[b]	CC[c]
1	Triângulo Simplificado	3	inteiro	[0,65535]	4	13
2	Resto	2	inteiro	[0,65535]	5	2
3	Produto	2	inteiro	[0,1023]	6	4
4	Busca Linear	1	inteiro	[0,16838]	5	4
5	Busca Binária	1	inteiro	[0,16838]	12	5
6	Valor do Meio	3	inteiro	[-32768,32768]	6	6
7	Triângulo	3	inteiro	[0,65535]	6	10

Tabela 4.1: Características dos SUTs

[a] Número de variáveis de entrada.

[b] Número de caminhos alvo escolhidos.

[c] Complexidade ciclomática de McCabe [27], medida utilizando a ferramenta Metrics[1].

O primeiro SUT classifica um triângulo em diversas categorias ("não triângulo", equilátero, isósceles e escaleno). O segundo SUT calcula o resto da divisão de dois números enquanto que o terceiro SUT encontra a multiplicação dos dois valores de entrada. No quarto e quinto SUTs temos os algoritmos de busca linear e binária. Ambos buscam um valor (chamado de "chave") em um vetor previamente inicializado aleatoriamente com valores do domínio. Na busca linear o vetor possui treze valores enquanto que o vetor da busca binária possui quarenta. Estas dimensões são as mesmas utilizadas no experimento de Abreu [58] e foram mantidas. O sexto SUT indica dentre os três valores de entrada qual é o "valor do meio". O último, assim como o primeiro, classifica um triângulo mas desta vez em mais categorias ("não triângulo", equilátero, isósceles, acutângulo, obtusângulo e reto).

4.2 Procedimento para Execução dos Algoritmos

Todos SUTs tiveram seus GFCs (Grafo de Fluxo de Controle) desenhados e a partir disso os códigos fontes foram instrumentados manualmente inserindo-se um método estático que concatenava um *char* em uma *string* a cada aresta do GFC coberta. Por exemplo,

[1] http://metrics.sourceforge.net/

após uma execução do SUT Valor do Meio (figura 3.1) que exercitasse o caminho 0-1-2-3-4-9, uma variável *string* estática[2] permaneceria com o valor "0,1,2,3,4,9"onde ',' é um separador. Analisando-se esta *string* é possível obter o caminho percorrido pela execução. A figura 4.1 apresenta a estrutura utilizada para se coletar as informações das execuções deste estudo empírico para os algoritmos GEOcan, GEOvar, GEOreal e geração aleatória.

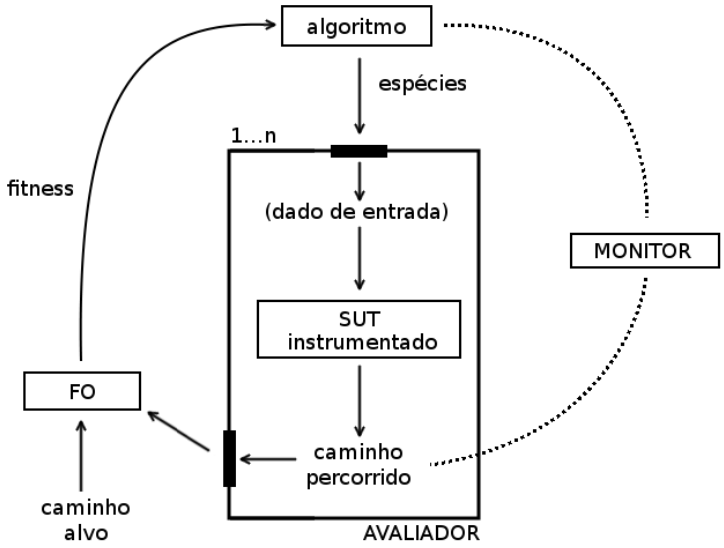


Figura 4.1: Estrutura do estudo empírico para GEOs e geração aleatória

Todo algoritmo utilizado, excetuando-se a geração aleatória, possui uma população de espécies que são dados de entrada para o SUT abordado. Este algoritmo deve entregar toda a população para o "módulo avaliador" para que cada espécie tenha seu valor de *fitness* calculado pela função objetivo. No caso da geração aleatória apenas o dado gerado naquela iteração é entregue para o "módulo avaliador".

Antes da função objetivo poder calcular o *fitness* de cada espécie, todos são executados sobre o SUT o que resulta em diversos caminhos percorridos, um para cada espécie (dado de entrada). Só então é que os resultados são passados para a função objetivo. Este passo é importante para que o função objetivo Bueno e Jino possa ser calculada.

A partir daí pode-se calcular o *fitness* utilizando-se o caminho percorrido, o caminho alvo e as informações das outras execuções. Este valor é retornado para o algoritmo que inicia uma nova iteração sobre a população. As iterações continuam até que o critério de parada seja atingido, que neste caso é um número máximo de avaliações da FO. Este número depende do SUT abordado.

Existe ainda o "módulo monitor" que observa o algoritmo em execução e os caminhos

[2]http://download.oracle.com/javase/tutorial/java/javaOO/initial.html - item *Initializing Fields*

percorridos. Caso o algoritmo atinja o número de avaliações da FO máximo o "módulo monitor" pára a execução deste estudo. O mesmo acontece caso o caminho alvo seja encontrado. Em ambas as situações as informações da execução são armazenadas e uma nova execução é iniciada até que se atinja o número de execuções determinada para aquele algoritmo/SUT.

Os algoritmos SGA e RCGA foram implementados de forma diferente dos outros algoritmos e isto é detalhado mais abaixo. Por este motivo a estrutura para coletar as informações das execuções foi um pouco diferente, mostrado na figura 4.2. Neste caso cada indivíduo era avaliada individualmente sem a presença das informações das outras espécies.

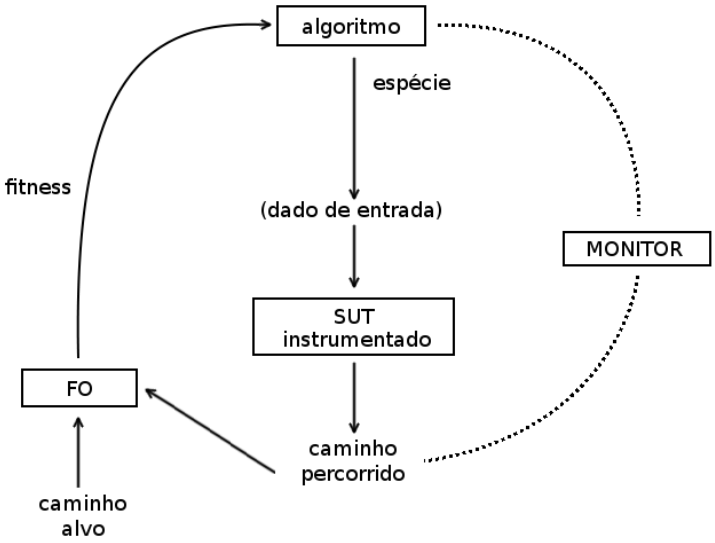


Figura 4.2: Estrutura do estudo empírico para SGA e RCGA

4.2.1 Comparação dos Resultados

Todos os resultados das execuções algoritmos/SUT foram comparados através do critério de porcentagem média de cobertura adquirida durante a geração de dados de teste. Este critério aponta qual é a porcentagem média de caminhos cobertos pelo algoritmo em relação ao número de avaliações da função objetivo. Os caminhos alvo de cada SUT são os pré determinados no trabalho de Abreu [58] e foram mantidos para comparação dos resultados.

Também foram comparados os resultados dos algoritmos trocando-se a função objetivo. Ao utilizar a função *Similarity* todos os algoritmos foram executados mas ao utilizar a

função de Bueno e Jino [48] somente os algoritmos GEO foram executados. A razão desta decisão está na característica inerente ao cálculo desta última FO. Ela necessita que todas as populações temporárias do GEO (dados de entrada dos testes) tenham sido executados sobre o SUT para então poder avaliar cada uma delas. A seção 3.2.2 explica como a FO de Bueno e Jino [48] precisa no segundo passo de cálculo de todas as execuções que desviaram do caminho alvo na mesma aresta. Como os algoritmos GEO foram implementados independentemente de uma biblioteca foi possível codificá-los para suprirem todas as informações para o cálculo da avaliação da FO para cada população temporária. Este já não foi o caso para os algoritmos SGA e RCGA. Ambos são baseados na biblioteca JGAP[3] (JGAP – *Java Generic Algorithms Package*) e a avaliação de cada indivíduo é feito sem que se possua informação dos outros inviabilizando o cálculo desta FO. Portanto nos gráficos de comparação através do critério de porcentagem média de cobertura adquirida para a função objetivo Bueno e Jino [48] apenas os algoritmos GEO e a geração aleatória serão mostrados.

Com o objetivo de melhor entender os resultados obtidos os gráficos da avaliação da função objetivo foram plotados dentro de um pequeno domínio de apenas cem valores. Com esta quantidade há uma melhor visualização dos gráficos obtidos. Não foi necessário utilizar todo o domínio original de cada SUT pois no pequeno domínio especificado todos os caminhos são exercitados.

4.3 Ajustando os Parâmetros dos Algoritmos

Os desempenhos de algoritmos baseados em metaheurísticas pode variar de forma significativa dependendo do problema abordado e da função objetivo utilizada [29]. Desta maneira experimentos foram executados para cada um dos SUTs de maneira a obter os melhores parâmetros para cada algoritmo.

Apesar desta pesquisa utilizar os mesmo algoritmos utilizados por Abreu [58] suas implementações (código fonte) são diferentes. Neste trabalho os algoritmos SGA e RCGA utilizaram uma biblioteca altamente difundida chamada JGAP enquanto que Abreu codificou estes algoritmos sem utilizar esta biblioteca. Os outros algoritmos também foram codificados de forma diferente com o intuito de melhorar o desempenho já que foram implementados em Java 1.5[4] enquanto que Abreu os fez em Java 1.4[5].

É importante mencionar que a biblioteca JGAP foi escolhida para ser utilizada pois é muito conhecida na comunidade de desenvolvimento Java. Isto facilita a réplica destes experimentos por qualquer outro estudo caso o código fonte deste trabalho não esteja

[3] http://jgap.sourceforge.net/
[4] http://www.java.sun.com
[5] http://www.java.sun.com

disponível. De fato, apesar de ter acesso ao código fonte utilizado na pesquisa de Abreu, este não era passível de ser evoluído para incluir a função objetivo Bueno e Jino nem o algoritmo GEOreal. Por este motivo todo o estudo empírico foi novamente codificado.

Os algoritmos estudados são: SGA, RCGA, GEOcan, GEOvar, GEOreal. A geração aleatória não necessita de nenhum estudo pois não necessita de parâmetro de ajuste. Nos algoritmos SGA e RCGA três são as variáveis ajustadas: probabilidade de mutação (p_m), probabilidade de recombinação (p_c) e tamanho da população (*popsize*). Nos algoritmos GEOcan, GEOvar, GEOreal apenas a variável τ foi ajustada. As condições de ajuste foram as mesmas de Abreu [58]:

- O algoritmo foi aplicado 20 vezes para cada combinação de parâmetros possível.
- O número máximo de avaliações da função objetivo foi limitada em 100.000.
- Para os algoritmos GEOcan, GEOvar, GEOreal o parâmetro τ variou em incrementos de 0,25 de 0 à 10.
- Para os algoritmos SGA e RCGA:
  - A população assumiu apenas os valores 100, 1000 ou 10000.
  - A probabilidade de recombinação variou de 0,6 a 1,0 com incrementos de 0,1.
  - A probabilidade de mutação variou de 0,001 a 0,0205 com incrementos de 0,0015.

Fica claro que o número combinações de parâmetros ajustáveis para os algoritmos GEOcan, GEOvar, GEOreal são menores que para os algoritmos SGA e RCGA. Nos primeiro o número é de 41 combinações enquanto que nos últimos temos 210. O tempo dispendido para encontrar as combinações foi muito elevado para os algoritmos SGA e RCGA.

O critério de escolha dos parâmetros que seriam utilizados nos experimentos foi o de cobertura total dos caminhos-alvo do SUT (mostrados na tabela 4.1 - coluna NTP). Para cada uma das 20 execuções com os parâmetros, uma contagem foi feita para o número de execuções que obtiveram 100% de cobertura do SUT. A combinação que obteve o maior número de cobertura total foi esco-lhido. Os parâmetros finais para cada algoritmo versus SUT são mostrados na tabela 4.2. Ambas as funções objetivo foram utilizadas mas isto não afetou os parâmetros escolhidos, por este motivo apenas uma tabela foi mostrada.

Diversos comentários podem ser feitos sobre os resultados do experimento de ajuste de parâmetros. No SUT triângulo simplificado os testes para o algoritmo SGA atingiram no melhor caso apenas 3% de cobertura total em três combinações. Todas as outras combinações atingiram menos que isso. Para o algoritmo RCGA todas as combinações não conseguiram atingir nenhuma cobertura total. Desta maneira qualquer combinação

#	**SGA**	**RCGA**	**GEOcan**	**GEOvar**	**GEOreal**
Triângulo Simplificado	popsize:10000 p_c:1.0 p_m:0.0025	popsize:10000 p_c:1.0 p_m:0.0025	τ:0.75	τ:0.75	τ:0.75
Resto	popsize:1000 p_c:0.6 p_m:0.006	popsize:100 p_c:0.6 p_m:0.002	τ:0.75	τ:0.75	τ:0.75
Produto	popsize:1000 p_c:0.9 p_m:0.0025	popsize:1000 p_c:0.9 p_m:0.0025	τ:0.75	τ:0.75	τ:0.75
Busca Linear	popsize:10000 p_c:0.7 p_m:0.001	popsize:10000 p_c:0.7 p_m:0.001	τ:0.25	τ:0.25	τ:0.25
Busca Binária	popsize:1000 p_c:0.7 p_m:0.001	popsize:1000 p_c:0.7 p_m:0.01	τ:0.75	τ:0.75	τ:0.75
Valor do Meio	popsize:100 p_c:0.8 p_m:0.015	popsize:100 p_c:0.8 p_m:0.015	τ:0.75	τ:0.75	τ:0.75
Triângulo	popsize:10000 p_c:0.9 p_m:0.001	popsize:10000 p_c:0.9 p_m:0.001	τ:0.75	τ:0.75	τ:0.75

Tabela 4.2: Parâmetros finais para os algoritmos

poderia ser usada pois em nenhuma combinação era melhor que a outra. Por este motivo os parâmetros do algoritmo RCGA estão iguais ao do SGA. Para o algoritmo GEOcan e GEOvar em nenhum valor de τ atingiu-se cobertura total. Para o algoritmo GEOreal todos os valores de τ atingiram 100% de cobertura total. Desta maneira o valor de 0,75 foi escolhido por ser um valor usualmente utilizado na execução deste algoritmo.

Praticamente os mesmos comentários podem ser feitos sobre o SUT triângulo complexo. Para nenhum valor dos parâmetros os algoritmos SGA, RCGA, GEOcan e GEOvar obtiveram cobertura total nas 20 execuções. Já o algoritmo GEOreal obteve 100% de cobertura total para todas as execuções. Os parâmetros do SGA e RCGA foram mantidos iguais aos do trabalho de Abreu [58]. Já os valores de τ foram padronizados em 0,75. O mesmo raciocínio foi aplicado para os SUTs busca linear, busca binária – que não obteve nenhuma cobertura total com nenhum valor dos parâmetros – e valor do meio – que obteve cobertura total em 100% dos casos. Apenas o SUT busca linear teve os valores de τ ajustado para 0,25 pois nestes casos os algoritmos GEO obtiveram cobertura total em cerca de 10% dos casos.

Nos SUTs restantes os parâmetros foram ajustados para os valores indicados na tabela. Esses foram os parâmetros nos quais o algoritmo obteve melhor número de coberturas total do problema.

Vemos claramente que muitos dos parâmetros são distintos do trabalho de Abreu [58]. Isso se deve a diferente implementação dos códigos de todos os algoritmos. Por este motivo este passo de ajuste dos parâmetros foi executado novamente.

4.4 Experimentos e Resultados

Cada combinação de algoritmo/SUT foi executado duzentas vezes em um laptop com processador Intel Core 2 Duo P8600 com 2,4GHz e 4GB de memória RAM. Os resultados foram apresentados mostrando gráficos de cobertura média dos caminhos pré-escolhidos em cada SUT para cada FO. Para melhor compreender esses resultados, gráficos do domínio de cada função objetivo para cada SUT foram plotados utilizando o programa *MatLab* 7.0 no sistema operacional *Windows Vista*.

A próxima subseção apresentará apenas a análise do SUT Triângulo Simplificado enquanto que a análise dos outros SUT está no apêndice A. Dentre todos os SUTs da tabela 4.1 o Triângulo Simplicado é o que possui maior complexidade ciclomática e a análise de suas execuções trouxe resultados interessantes. Por este motivo ele foi escolhido para ser apresentado aqui enquanto que os outros SUTs, que também possuem informações relevantes mas muito próximas do Triângulo Simplificado, foram deslocadas para o apêndice.

4.4.1 Triângulo Simplificado

O SUT triângulo simplificado é o primeiro a ser analisado. Inicialmente o GFC é montado para que a partir dele possamos escolher os caminhos que serão escolhidos como alvo. A partir da figura 4.3 podemos observar quatro caminhos independentes: 0-1-2-8-9, 0-1-2-3-4-9, 0-1-2-3-5-6-9, 0-1-2-3-5-7-9. Estes são exatamente os caminhos alvo deste problema.

```
0 private TriangleType simpleTriangle(final int x, final int y, final int z) {
1     TriangleType type;
2     if ((x + y > z) && (y + z > x) && (z + x > y)) {
3           if ((x != y) && (y != z) && (z != x)) {
4                 type = TriangleType.ESCALENO;
            } else {
5                 if (((x == y) && (y != z)) || ((y == z) && (z != x)) || ((z == x) && (x != y))) {
6                       type = TriangleType.ISOSCELES;
                  } else {
7                       type = TriangleType.EQUILATERO;
                  }
            }
      } else {
8           type = TriangleType.NO_TRIANGLE;
      }
9     return type;
  }
```

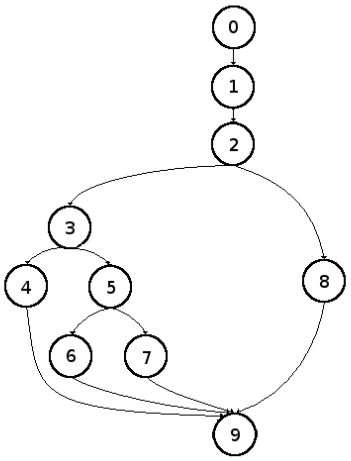


Figura 4.3: CFG - Triângulo Simplificado

Cobertura Média dos Algoritmos

A condição de parada foi de no máximo 100.000 avaliações da função objetivo. Na figura 4.4 podemos perceber que praticamente todos os algoritmos alcançam logo nas primeiras execuções 50% de cobertura que são os dois primeiros caminhos: para "não triângulo" e escaleno. A partir daí os algoritmos têm dificuldade para encontrar os dois próximos caminhos que são os de triângulo isósceles e equilátero (nesta sequência). Como pode ser observado no estudo do domínio da *Similarity* na figura 4.7 poucos conjuntos de pontos conduzem o SUT pelo caminho do triângulo isósceles e apenas um leva ao triângulo equilátero. Excetuando-se o algoritmo GEOreal todos tiveram dificuldade em encontrar estes dois últimos caminhos. Percebe-se que os algoritmos de SGA, GEOcan, GEOvar e a geração aleatória acabam por encontrar o caminho do triângulo isósceles até as 100.000 avaliações pois a cobertura varia entre 50% e 75%, mas não conseguem o de triângulo equilátero. Apenas os algoritmos que utilizam codificação real - GEOreal e RCGA - conseguiram a cobertura total do caminhos.

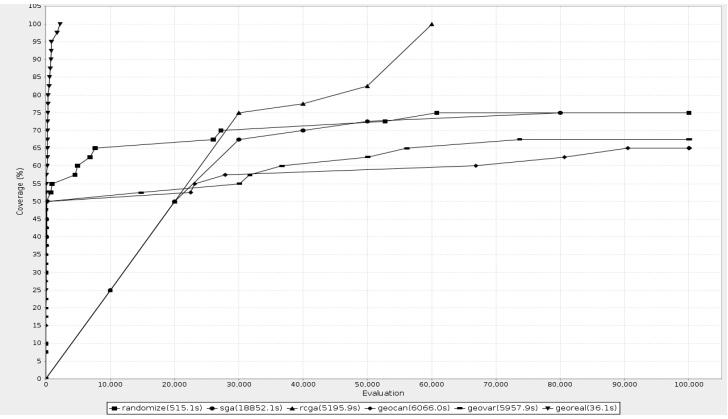


Figura 4.4: SUT Triângulo Simplificado (100.000 avaliações) – cobertura média dos algoritmos usando FO *Similarity*

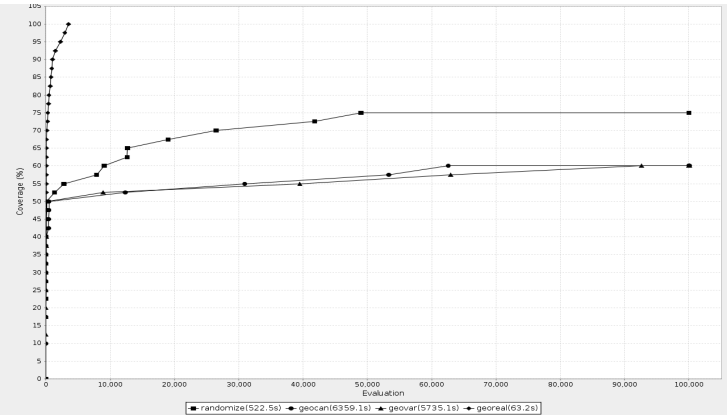


Figura 4.5: SUT Triângulo Simplificado (100.000 avaliações) – cobertura média dos algoritmos usando FO Bueno e Jino

Ao analisar a figura 4.5 observa-se que o algoritmo GEOreal obteve cobertura total rapidamente enquanto que os algoritmos GEOcan, GEOvar e a geração aleatória cobrem facilmente os dois primeiros caminhos (“não triângulo” e escaleno), pois grande parte do domínio satisfaz estes requisitos de teste, e o terceiro (isósceles) é, em algum momento, coberto.

Avaliação da Função Objetivo *Similarity*

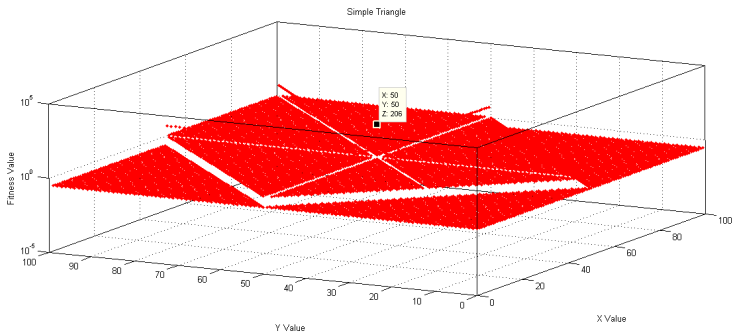


Figura 4.6: Avaliação da FO *Similarity* - Triângulo Simplificado

As duas figuras 4.6 e 4.7 apresentam a avaliação da função objetivo. Este SUT, assim como os SUTs Triângulo e Valor do Meio, possuem três variáveis de entrada. Para plotar o gráfico da avaliação da função objetivo uma das entradas foi fixada enquanto que as outras duas foram variadas. Desta maneira obtém-se dois eixos (X e Y) sendo que o terceiro (eixo Z) é o valor da avaliação da FO em escala logarítmica.

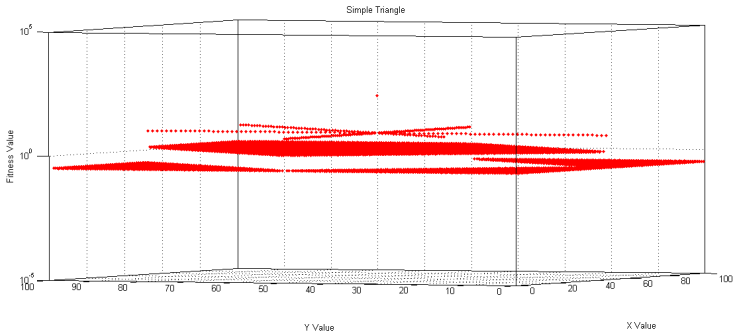


Figura 4.7: Avaliação da FO *Similarity* - Triângulo Simplificado

As variáveis de entrada X, Y e Z são os três lados do triângulo. O domínio para as duas primeiras variáveis foi [0,100]. A terceira variável (Z) foi escolhida para ter seu valor fixo em cinquenta (exatamente o meio do domínio plotado) permitindo assim uma melhor visualização do gráfico de avaliação da FO.

Ao observar a figura 4.6 percebemos claramente quatro planos distintos. O plano mais baixo é o que contém o maior número de pontos. O segundo plano (de baixo para cima) também contém uma boa quantidade de pontos enquanto que o terceiro plano se destaca por se compor de três retas bem distintas. Por fim temos o plano mais alto que é composto

por apenas um ponto.

Os planos refletem os quatro caminhos escolhidos para este SUT. Cada caminho leva a um tipo de triângulo: "não triângulo", escaleno, isósceles e equilátero. O plano mais baixo é composto por todos os conjuntos de pontos (X,Y,Z=50) que não formam um triângulo pois a soma de dois lados acaba por ser menor que o terceiro lado. Um exemplo seria X = 10, Y = 70, Z =50. O segundo plano é composto pelos pontos (X,Y,Z) que formam um triângulo escaleno. No terceiro plano temos as três retas que possuem os pontos para um triângulo isósceles. Perceba que uma das retas é a reta onde todos os pontos X e Y tem o mesmo valor. As outras duas retas são aquelas onde X vale sempre 50 ou Y vale sempre 50, ou seja, o mesmo valor de Z (que foi fixado em 50). Finalmente temos o último plano formado pelo ponto X=50, Y=50, Z=50, visível na figura 4.7 e com seus valores X, Y e Z destacados na figura 4.6. O valor da FO para este ponto é duzentos e seis, que é o maior valor da FO para este problema.

Ao executar o pequeno programa que gerou os dados dos gráficos 4.6, 4.7, 4.8 e 4.9 (variando os valores de X e Y e armazenando o valor avaliado pela FO) é necessário definir um caminho alvo já que para cada conjunto X, Y, Z introduzidos no SUT um caminho percorrido é gerado e só é possível calcular o valor da avaliação da FO através da comparação entra a distância entre um caminho alvo e um caminho percorrido. O caminho alvo escolhido neste SUT foi o mais difícil de ser coberto (pois do domínio de entrada deve-se ter X=Y=Z): o do triângulo equilátero. Como a FO *Similarity* retorna o maior valor para o caminho percorrido que é igual ao caminho alvo percebemos que o conjunto X,Y, Z para o triângulo equilátero possui o maior valor de avaliação da FO. Isso, de fato, reflete o esperado: o caminho alvo utilizado foi o caminho para o triângulo equilátero (X=Y=Z). O ponto (X=50, Y=50, Z=50) é o que possui maior avaliação da FO.

Avaliação da Função Objetivo Bueno e Jino

Seguiu-se o mesmo raciocínio descrito na seção 4.4.1. A partir da figura 4.8 e 4.9 percebe-se claramente três curvas distintas que refletem os pares (X,Y,Z) que levam a execução dos três primeiros caminhos possíveis ("não triângulo", escaleno e isósceles). No topo vemos o ponto (50,50,50) referente ao caminho do triângulo equilátero.

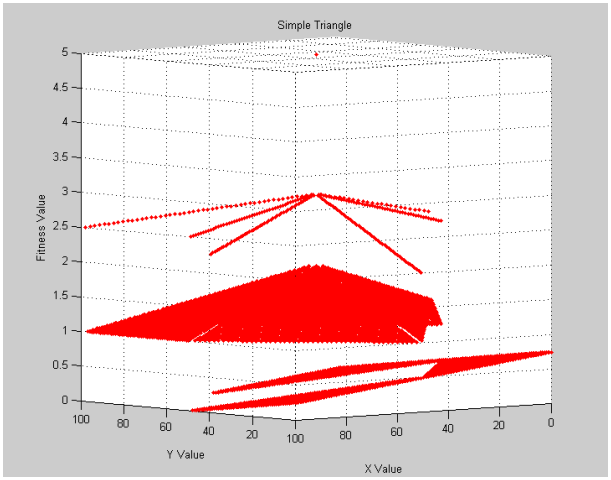


Figura 4.8: Avaliação da FO Bueno e Jino - Triângulo Simplificado

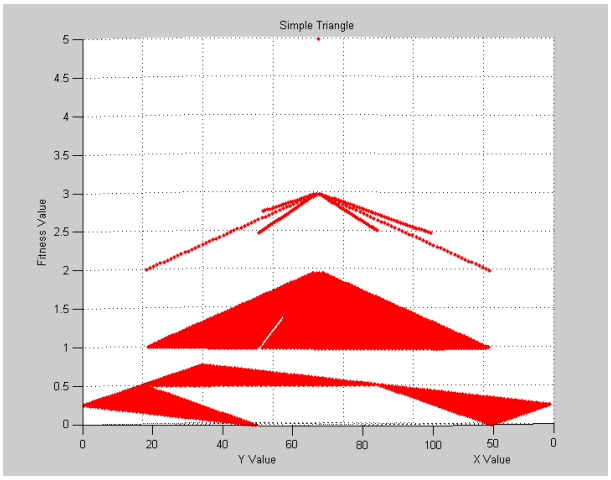


Figura 4.9: Avaliação da FO Bueno e Jino - Triângulo Simplificado

A curva inferior é composta pelo maior número de pontos do domínio – são os que levam à execução do caminho "não triângulo". Em seguida (de baixo para cima) temos os pontos que levam a execução do caminho do triângulo escaleno. Após isso verificamos

três linhas com os pontos do triângulo isósceles e finalmente o único ponto referente ao caminho alvo: o do triângulo equilátero. Este ponto é de fato avaliado com maior valor pela FO.

Em comparação à avaliação da FO *Similarity* vemos grande diferença: todas os planos são formados por curvas inclinadas na direção do ponto de maior avaliação (50,50,50). Esta característica da função objetivo conduz melhor o algortimo na busca pela solução do que a presença de platos (visto na *Similarity*).

4.5 Discussão

Na tabela 4.3 são apresentados os resultados das execuções de cada algoritmo para cada SUT para a função objetivo *Similarity*. São mostrados o número médio de avaliações da função objetivo. Percebe-se bons resultados por parte do algoritmo GEOreal em praticamente todos os SUTs. Verifica-se que nos problemas com baixa complexidade – vide tabela 4.1 – a geração aleatória obtém melhor desempenho mas nos de maior complexidade o algoritmo GEOreal executa menos avaliações da FO. O fato da geração aleatória obter melhor desempenho nos SUTs de baixa complexidade está associado ao fato de que nestes problemas boa parte do domínio de entrada é solução para a busca. A geração aleatória consegue, com poucas iterações, encontrar um ponto desse domínio enquanto que um algoritmo evolutivo, dependendo de onde ele inicie, pode levar mais iterações para encontrar a solução já que a caminhada na direção do domínio de soluções é feita de forma progressiva, o que toma mais iterações que a geração aleatória.

#	**Random**	**SGA**	**RCGA**	**GEOcan**	**GEOvar**	**GEOreal**
Triângulo Simplificado	100.000(0)	99.000(3.456)	58.000(4.000)	100.000(0)	100.000(0)	**872(475)**
Resto	**143(76)**	55.000(13.725)	55.300(16.119)	1.852(1.422)	861(565)	225(107)
Produto	479(405)	29.700(18.595)	100.000(0)	3.450(3.457)	2.138(2.199)	**128(99)**
Busca Linear	89.303(32.252)	400.000(0)	268.000(135.114)	210.669(88.191)	214.833(68.747)	**6.305(3.202)**
Busca Binária	172.640(40.220)	400.000(0)	400.000(0)	218.427(65.684)	386.510(31.835)	**77.652(107.950)**
Valor do Meio	**35(11)**	600(0)	600(0)	3.100(3.715)	422(245)	130(56)
Triângulo	100.000(0)	100.000(0)	100.000(0)	100.000(0)	100.000(0)	100.000(0)

Tabela 4.3: Número de avaliações médio da FO por cada algoritmo em cada SUT utilizando a função objetivo *Similarity* (desvio padrão entre parenteses)

Observa-se em alguns campos das tabelas os valores 100.000 (0) e 400.000 (0). Isso significa que o algoritmo não conseguiu encontrar todos os caminhos alvo até o limite máximo de avaliações da FO em todas as duzentas execuções. Por este motivo o número

médio de avaliações mostrou este limite. A partir desta informação pode-se entender que de fato nenhum algoritmo conseguiu cobrir todos os caminhos alvo do SUT Triângulo. Ao verificar mais detalhadamente o motivo percebeu-se que alguns algoritmos – RCGA e GEOreal – cobriam todos os caminhos menos o que exercita o "triângulo retângulo". Já os algoritmos restantes nem conseguiram cobrir caminhos mais simples. De fato o SUT Triângulo possui caminhos complicados de serem cobertos como o de "triângulo retângulo" e "triângulo equilátero". Estes são exercitados apenas por poucos valores do domínio de entrada. Isso exige que tanto o algoritmo como a FO sejam capazes de guiar a busca de forma efetiva, coisa que não ocorreu com este SUT por nenhum algoritmo.

A tabela 4.4 apresenta os resultados para a função objetivo Bueno e Jino. Apenas a geração aleatória e os algoritmos GEOcan, GEOvar e GEOreal foram executados por motivo já explicado anteriormente.

Ao analisar as duas tabelas com os resultados observa-se que para cada SUT o mesmo algoritmo que obteve o melhor desempenho foi mantido independentemente da FO utilizada. Também é possível perceber uma melhora (diminuição) no número médio de avaliações da FO com o uso da Bueno e Jino. A melhor variação (43,3%) ocorreu na combinação GEOcan/Resto. Outro detalhe perceptível é o enorme desvio padrão na combinação GEOreal/Busca Binária (independente da FO). Por este ser o SUT com mais número de caminhos – vide tabela 4.1 – podemos concluir que o algoritmo GEOreal teve grande variação no número de avaliações para cobrir cada caminho individualmente e isso se acumulou por todos os doze caminhos do SUT.

#	**Random**	**GEOcan**	**GEOvar**	**GEOreal**
Triângulo Simplificado	100.000(0)	100.000(0)	100.000(0)	**818(649)**
Resto	**134(78)**	1.049(604)	859(525)	218(111)
Produto	489(505)	2.375(2.286)	1.921(1.707)	**111(64)**
Busca Linear	85.303(35.042)	191.022(92.278)	182.936(64.181)	**5.072(3.637)**
Busca Binária	180.094(39.137)	215.486(61.800)	364.362(52.964)	**76.252(108.737)**
Valor do Meio	**36(14)**	2.482(2.324)	379(267)	125(66)
Triângulo	100.000(0)	100.000(0)	100.000(0)	100.000(0)

Tabela 4.4: Número de avaliações médio da FO por cada algoritmo em cada SUT utilizando a função objetivo Bueno e Jino (desvio padrão entre parênteses)

Ao analisar todos os resultados obtidos com este estudo empírico pode-se perceber que o algoritmo GEOreal combinado com a FO Bueno e Jino obtém o melhor desempenho nos SUTs considerados. Apesar dos problemas abordados serem simples os resultados trazem boas expectivas quanto ao uso do algoritmo GEO em aplicações reais. Por este motivo um protótipo, apresentado no próximo capítulo, foi desenvolvido para aplicar estes conhecimentos.

Capítulo 5

Protótipo JET

O protótipo JET[1] (JET – *Java Evolutionary Tester*) foi desenvolvido com o objetivo de aplicar todos os conceitos envolvidos neste trabalho. Dessa maneira foi possível comprovar a aplicabilidade deste estudo e produzir uma espécie de "prova de conceito". Além disso o JET foi comparado com outras ferramentas já existentes para assim obter informações sobre o seu desempenho ao atacar o problema de geração de dados de teste estrutural.

Este capítulo apresenta uma extensa pesquisa por ferramentas disponíveis no mercado e em seguida como foi estabelecido o teste padronizado – benchmark – para avaliar o desempenho das ferramentas. Por fim detalha o funcionamento do JET, suas limitações e os resultados da comparação.

5.1 Sobre as Ferramentas

Atualmente existem muitas ferramentas para geração de dados de teste com acesso ao código-fonte Java. Neste trabalho apenas as gratuitas e disponíveis na internet foram pesquisadas. Esta seção apresenta as ferramentas encontradas e descreve resumidamente o seu funcionamento baseada em suas publicações e documentação.

5.1.1 eCrash

A ferramenta eCrash[2] foi desenvolvida pelos estudantes da Universidade de Coimbra[3] e atualmente é instalada como um plugin para o IDE Eclipse[4]. A primeira publicação [28] que descreve esta ferramenta é de Setembro de 2007 e desde então o eCrash tem sido

[1] Código fonte disponível em https://github.com/buzzo/jet
[2] http://mse-gunners.dei.uc.pt/ecrash/help
[3] http://www.uc.pt/fctuc/dei/
[4] http://www.eclipse.org

atualizado e melhorado.

O eCrash foca na geração de dados de teste tendo acesso ao código fonte de classes Java[5]. O método de otimização utilizado para a geração é conhecido como Programação Genética com Tipagem Forte (STGP – *Strongly-Typed Genetic Programming*).

5.1.2 EvoTest

O EvoTest[6] é um projeto multidisciplinar desenvolvido inicialmente no Instituto Tecnológico de Informática[7] em Valência – Espanha. Este projeto tem como objetivo combinar as técnicas de adaptação evolutiva com a engenharia de *software* e encontrar soluções para problemas de testes de *software*.

Apesar do site na internet apresentar diversas informações e publicações [57, 31, 41], nenhuma é específica sobre uma ferramenta. São pesquisas abordando a utilização de múltiplas funções objetivo, algoritmos de busca global, testes estruturais e relatórios sobre um *framework* chamado AETF (AETF – *Automated Evolutionary Testing Framework*).

Nenhuma ferramenta é disponibilizada no site do EvoTest na internet. Apesar de enviar *e-mail* para os responsáveis requisitando este *software* – dito gratuito para pesquisa acadêmica – nenhuma resposta foi dada. Apesar disto a menção desta ferramenta é válida, dada a grande quantidade de publicações associadas ao nome EvoTest.

5.1.3 eToc

O eToc[8] foi desenvolvido em Trento – Itália e possui uma publicação [51] que demonstra seu funcionamento. A ferramenta, apesar de disponível na internet, tem sua última atualização datada em Abril de 2004 o que aponta para uma descontinuidade do seu desenvolvimento.

O objetivo do eToc é cobrir todos os condicionais do CUT (CUT - *Class Under Test*) abordado dentro de um tempo limite determinado pelo usuário. Para esta tarefa um GA com operadores genéticos modificados é utilizado. Esta especialização é necessária pois os cromossomos da popu-lação do GA são compostos de combinações de chamadas ao métodos do CUT. Portanto o conceito de seleção, recombinação e mutação foi mantido mas a implementação destes foi especializada para poder atacar o problema de geração de dados de teste.

[5] http://www.java.sun.com/
[6] http://evotest.iti.upv.es/
[7] http://squac.iti.upv.es/
[8] http://star.fbk.eu/etoc/

5.1.4 testful

O testful[9] foi proposto e desenvolvido pela Politécnica de Milão[10] – Itália. Dentre todas as ferramentas esta é a que possui melhor documentação e um grande número de publicações [33, 32, 9]. Em um dos documentos disponíveis em seu site, um estudo comparativo foi feito entre o desempenho do testful e de outras ferramentas. Esta foi uma boa fonte de referência para esta pesquisa.

A busca por dados de teste é executada em duas fases distintas. Na primeira executa-se, por um pequeno período de tempo, uma busca aleatória para iniciar diversas populações com baixa qualidade. Na segunda fase uma porcentagem dessas populações – as melhores adaptadas – são evoluídas utilizando-se um algoritmo híbrido. É usado o algoritmo genético [33] combinado com o *Hill Climbing* [41]. O primeiro busca as sequências das chamadas dos métodos, para tentar colocar o CUT abordado em um determinado estado para ser testado, enquanto que o segundo algoritmo é executado localmente, dentro de cada método. O objetivo é conseguir cobrir o maior número de condicionais no tempo limite escolhido pelo usuário.

5.1.5 jAutoTest

A ferramenta jAutoTest possui diversas referencias [8] na internet mas o seu site oficial[11] não possui qualquer código-fonte ou documentação. Esta ferramenta foi utilizada no estudo da testful [33] e por isso é candidata a ser avaliada.

5.1.6 randoop

O randoop[12] não é uma ferramenta baseada em busca evolutiva mas um *framework* de geração aleatória de dados de teste. Possui duas versões: uma para Java, feita por um grupo do MIT – Estados Unidos, e outra para .NET, feito pela *Microsoft*. Esta ferramenta foi escolhida por ser um *software* conhecido de geração aleatória. Desta maneira não foi necessário desenvolver um *software* equivalente somente para fazer a comparação com o JET.

A ferramenta randoop possui uma boa documentação e código-fonte disponível. Também possui alguns desenvolvedores que estão continuamente evoluindo o *software*. Existem publicações [16, 15] mostrando que o randoop tem bons resultados em encontrar falhas em diversos programas. O único detalhe que deve ser considerado ao utilizá-lo

[9] http://code.google.com/p/testful/
[10] http://www.english.polimi.it/
[11] http://sourceforge.net/projects/jautotest/
[12] http://code.google.com/p/randoop/

é o número de parâmetros que podem ser ajustados. Por exemplo, pode-se configurar o tempo de execução, algumas variáveis da geração aleatória, restrições quanto a quantidade de dados de teste gerado, etc. Existe uma configuração padrão mas geralmente algum parâmetro precisa ser ajustado.

5.2 Protótipo JET

O objetivo do JET é gerar casos de teste para métodos de classes Java utilizando o modelo de busca evolutiva. Internamente ao protótipo, a geração dos casos de teste é tomada como um problema de busca que é trabalhado pelas diversas implementações do algoritmo evolutivo GEO – GEOcan, GEOreal, GEOint.

5.2.1 Requisitos

O protótipo JET tem como objetivo gerar dados de teste para cobrir um determinado número de caminhos pré selecionados pelo usuário de um método Java. Desta maneira este protótipo tem como primeiro requisito a entrada, por parte do usuário, dos caminhos que devem ser cobertos. Por este motivo outro requisito necessário por parte do protótipo é a geração do CFG do método que será testado. Somente desta maneira será possível ao usuário a seleção dos caminhos do MUT (MUT - *Method Under Test*).

É necessário que o JET tenha acesso ao código do MUT para a geração de dados de teste. Não deve ser necessário, por sua vez, acesso ao código fonte do programa Java que será testado mas apenas ao *bytecode* pois isto facilita o teste de programas os quais o testador não possui os arquivos fonte. Outro requisito interessante do JET é a instrumentação do código *bytecode* em tempo de *runtime*. Isto significa que em apenas uma execução tanto a instrumentação do código quanto a execução da busca pelos dados de entrada são feitos. Este diferencial não é observado nas ferramentas já existentes que necessitam de um pré-passo de instrumentação e então outra (nova execução) para a busca. Isto torna mais simples a utilização do JET pelo usuário e facilita a construção de uma futura GUI (GUI - *Graphic User Interface*) sobre a ferramenta.

5.2.2 Funcionamento

Para explicar o funcionamento do protótipo JET a figura 5.1 apresenta sua estrutura interna e os dois modos de execução possíveis.

O JET é composto por três blocos: o instrumentador *bytecode*, um pacote de classes chamado de *jet-inject* e o código que realiza a busca utilizando o algoritmo GEO.

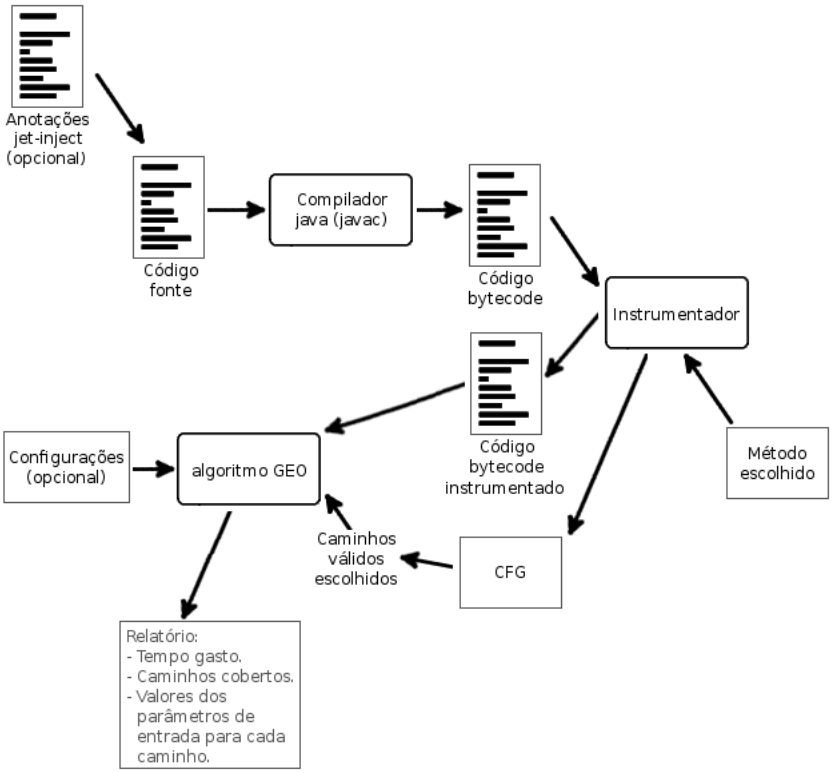


Figura 5.1: Estrutura interna do JET

O protótipo pode ser executado de duas maneiras distintas. Na primeira o usuário entra com a classe Java já compilada junto com a assinatura – nome, número e tipos dos parâmetros – do método que será atacado. A partir disto o instrumentador desenha o CFG do método e o retorna para o usuário. Desta maneira ele pode escolher quais caminhos devem ser cobertos. O segundo modo de executar o JET recebe como entrada os mesmos parâmetros explicados no primeiro modo de execução, adicionado dos caminhos a serem cobertos. Neste modo o protótipo já faz a busca pelos dados de teste para cobrir os caminhos e retorna os valores dos parâmetros de entrada para os casos de teste.

Independente do modo de execução escolhido o instrumentador *bytecode* tem papel importante no processo de montar o CFG para a escolha dos caminhos pelo usuário ou para a coleta de dados da execução durante a busca. Muitos programas de instrumentação *bytecode* foram pesquisados mas o que melhor se adequou ao requisitos deste projeto foi o *framework* Soot[13]. Ele disponibiliza uma API que facilita a montagem do CFG, além da instrumentação para a obtenção dos dados de uma execução do método. Os outros programas ou não permitiam acesso à instrumentação interna de um método – apenas à

[13]http://www.sable.mcgill.ca/soot/

chamada e término da execução de um método – ou eram antigos e não possuíam mais suporte.

O segundo bloco, chamado de *jet-inject*, tem como objetivo disponibilizar classes *Annotations*[14] para serem colocadas sobre os métodos que serão testados. Essas anotações têm como objetivo fixar o valor de entrada de um ou mais parâmetros. Este pacote é fundamental para que o protótipo possa abordar métodos que possuem tipos de parâmetro Java ainda não implementados pelo JET. Um exemplo é o tipo *string*. Em um método que possui um parâmetro *string* basta anotá-lo usando as classes do pacote *jet-inject* indicando um valor fixo que será sempre usado neste parâmetro em todas as execuções.

```
@IntArrayParameter(index = 1, value = { 1, 2, 3, 4, 5, 6 })
public boolean run(final int[] v, final int key) {
    boolean found = false;
    int i = 0;
.....
}


@StringParameter(index = 3, value = "message")
public static int addAndCheck(int a, int b, String msg) {
    int ret;
.....
}
```

Figura 5.2: Exemplos de utilização do pacote *jet-inject*

Atualmente o protótipo JET apenas consegue gerar dados de teste para métodos que possuem parâmetros do tipo *int* (inteiro), *double* e *float* (não gera para os *array* destes). Caso o método abordado possua tipos ainda não implementados o pacote *jet-inject* deve ser usado sobre esses parâmetros para que então o método possa ser abordado. Um detalhe importante ao se utilizar o pacote *jet-inject* é que o usuário precisa utilizá-lo sobre o código-fonte, ou seja, a utilização desta facilidade acaba por ser intrusiva. Isto é diferente de uma abordagem na qual o testador entra diretamente com o arquivo compilado sem qualquer modificação no código-fonte. Apesar disto, assim que todos os tipos Java forem implementados no JET o *jet-inject* se tornará obsoleto.

Por fim o terceiro bloco é responsável por executar as implementações do algoritmo GEO sobre o problema – o método abordado. É este bloco que determina qual implementação do GEO será utilizada dependendo do tipo dos parâmetros do método – por exemplo, utilizar o GEOint quando existem parâmetros inteiros. Além disso, é responsável por executar os dados de entrada do algoritmo sobre o método, ler o caminho percorrido e entregar todas as informações necessárias para a função objetivo que retorna o valor de

[14] http://download.oracle.com/javase/1.5.0/docs/guide/language/annotations.html

fitness para o algoritmo.

Assim como no instrumentador, um *framework* foi pesquisado para as implementações dos algoritmos GEO. Isso facilita a codificação e garante a utilização de pacotes conhecidos pela comunidade. O candidato escolhido foi o ECJ[15] [55], um *framework* público, atualizado e com suporte. Através do ECJ temos a execução do algoritmo sobre o problema – método da classe Java – guiado pela função objetivo com a responsabilidade de cobrir cada um dos caminhos selecionados pelo testador.

A única funcionalidade que o ECJ não implementa é a execução *multi-thread* onde cada *thread* seja responsável por todo ciclo do algoritmo evolutivo. Isto significa que não é possível tentar cobrir, em cada *thread* individualmente, um caminho selecionado pelo testador. Desta maneira seria possível iniciar diversas *threads* em paralelo, cada uma com a responsabilidade de cobrir um caminho.

Por este motivo o protótipo JET utilizou-se de manipulação de diversos *class-loaders*[16] para conseguir criar múltiplas *threads*, cada uma utilizando-se do ECJ para atacar o problema e cobrir um caminho alvo escolhido. Desta forma a busca por cobrir todos os caminhos alvos ocorre de forma paralela e assim obtém-se um melhor desempenho.

5.2.3 Limitações

Existem algumas limitações inerentes ao desenvolvimento do protótipo JET. Uma delas é a maneira como os SUTs são abordados. A geração de dados de teste é feita apenas sobre os métodos de uma única classe Java e não sobre diversos métodos de diversas classes. Isto se deve ao fato de que a proposta deste trabalho é de cobertura de caminhos pré selecionados de um método e esta abordagem não contempla a utilização de diversas classes.

A seleção dos SUTs, utilizados na comparação de desempenho das diversas ferramentas, foi impactada por esta limitação. Todos as classes escolhidas possuem apenas um método interno para ser executado sendo que este não depende de qualquer estado previamente preparado da classe. Ou seja, o método deve trabalhar exclusivamente com os dados de entrada (parâmetros) fornecidos e retornar algum resultado.

Outra limitação do protótipo é que o SUT é reiniciado em cada iteração do algoritmo evolutivo. Isto significa que uma nova instância da classe executada é criada para cada iteração do algoritmo GEO (quando uma população temporária é escolhida para ser a nova população corrente). Esta abordagem acaba por descartar qualquer estado previamente alcançado pela instância e é conhecida como *stateless*. Quando uma mesma instância de uma classe é utilizada por diversas iterações, mantendo-se assim o estado interno desta

[15] http://cs.gmu.edu/~eclab/projects/ecj/

[16] http://download.oracle.com/javase/1.4.2/docs/api/java/lang/ClassLoader.html

instância para cada iteração, temos a abordagem *stateful*.

Boa parte das ferramentas pesquisadas explicitamente dizem em sua documentação que utilizam a abordagem *stateful* enquanto que as demais não mencionam a abordagem utilizada. Isto não afetou a seleção de quais ferramentas seriam comparadas com o JET dado que a primeira limitação – trabalhar apenas com métodos – implica que mesmo que uma ferramenta utilize a abordagem *stateful* sobre a classe não há estado mantido internamente na instância.

Por causa de todas as limitações mencionadas, o JET não se encaixa na categoria de ferramenta. Apesar disto muitas das funcionalidades necessárias para uma ferramenta foram implementadas, como: instrumentação do *bytecode*, execução da busca através de diversas implementações de um algoritmo evolutivo, implementação de uma função objetivo, etc. No item 6.2 é explicado quais funcionalidades faltam ser implementadas em um trabalho futuro para que o JET seja considerada uma ferramenta.

5.3 Testes Padronizados - *Benchmark*

Com o propósito de avaliar as ferramentas e o protótipo JET, um conjunto de CUTs foi escolhido e uma estrutura para a coleta de dados foi montada para que todos os resultados fossem comparados de forma igualitária. O critério de seleção dos CUTs levou em consideração as limitações do JET, ou seja, o fato do protótipo apenas trabalhar com métodos. Portanto todas as classes selecionadas possuem apenas um método. Outro critério utilizado foi a de que os métodos deveriam possuir diversos tipos de parâmetros (inteiros, *double*, *arrays*, *string*, etc). Os parâmetros ainda não implementados pelo JET foram fixados com valores pré-determinados utilizando o pacote *jet-inject*.

Nove foram os CUTs escolhidos para os testes. São os sete problemas listados na tabela 5.1 mais dois problemas chamados *ArrayPartition* [42] e *MathUtils*[17]. Destes últimos dois, o primeiro é utilizado no estudo empírico da ferramenta testful [33] e foi escolhido por já haverem dados desta ferramenta para comparação. O CUT *MathUtils* foi escolhido por ser uma classe conhecida e distribuída pela fundação Apache[18].

O CUT *ArrayPartition* é um trecho de código do *QuickSort*, um algoritmo de ordenação extremamente conhecido e utilizado. Este código divide um *array* em duas partes através de um pivô (inicialmente o primeiro elemento do *array*) movendo os elementos menores para antes da posição do pivô e os maiores para depois deste. Idealmente isto deixará o pivô no meio destes dois grupos sendo que cada um deles é trabalhado pela mesma estratégia de partição.

O CUT *MathUtils* possui este nome pois é o nome da classe presente no pacote *apache-*

[17] http://commons.apache.org/math/
[18] http://commons.apache.org/math/

#	**CUT**
1	Triângulo Simplificado
2	Resto
3	Produto
4	Busca Linear
5	Busca Binária
6	Valor do Meio
7	Triângulo
8	*ArrayPartition*
9	*MathUtils*

Tabela 5.1: CUTs utilizados no teste padronizado

commons-math[19]. Esta classe recebe como parâmetro dois inteiros e uma *string*. Os inteiros, positivos ou negativos, são somados e o valor retornado, caso não haja *overflow* – a soma dos valores ultrapasse o valor Java inteiro máximo (2147483647) ou mínimo (-2147483648). Do contrário uma mensagem (parâmetro *string*) é lançada em uma exceção.

Cada ferramenta foi executada 200 vezes sobre cada um dos CUTs. Foi contabilizado o número de vezes que cada combinação CUT/ferramenta conseguiu cobrir todos os condicionais do método abordado. Nas execuções em que houve sucesso em atingir este objetivo contabilizou-se o tempo em milissegundos para esta tarefa. Portanto um dos indicadores de desempenho foi obtido contabilizando o tempo médio gasto nas execuções que obti-veram sucesso em cobrir totalmente o CUT. O outro indicador foi o número de sucessos em cobrir totalmente o CUT (graficos 5.4, 5.5 e 5.6)

A estrutura mostrada na figura 5.3 foi utilizada para a coleta de dados. Cada ferramenta foi executada recebendo como entrada o código *bytecode* de cada CUT. Este código foi previamente instrumentado manualmente para que quando cada nó do CFG do CUT fosse percorrido, uma impressão na tela (*System.out.println*("<número do nó CFG>")) mostrava um número referente ao nó executado. Um código "monitor" era responsável por ler estas impressões e contabilizar o tempo desde o início da execução até todos os nós do CFG serem percorridos. Quando isto ocorria o tempo dispendido no processo era armazenado. Também era responsabilidade do "monitor" limpar qualquer recurso e reiniciar a execução da ferramenta.

[19]http://commons.apache.org/math/

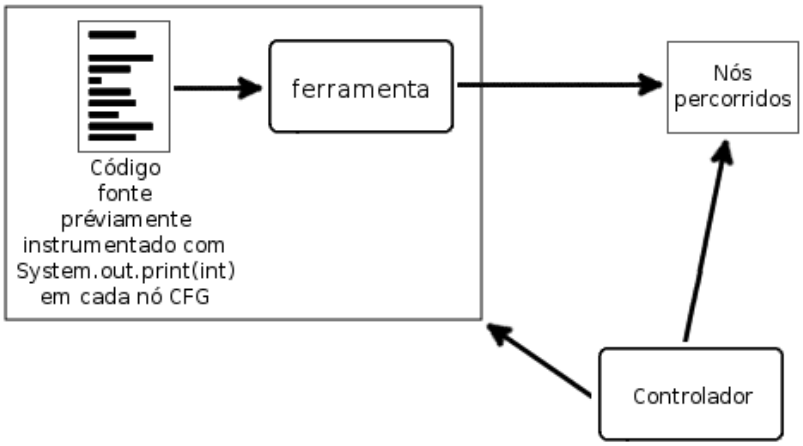


Figura 5.3: Estrutura para leitura dos dados das execuções das ferramentas

5.4 Seleção das Ferramentas

Dentre toda as ferramentas pesquisadas e mencionadas na seção 5.1 apenas algumas puderam ser comparadas com o JET devido a algumas restrições. Para explicar estas restrições a próxima seção menciona detalhes das implementações das ferramentas pesquisadas. Na seção seguinte há uma descrição de quais ferramentas foram selecionadas e o motivo disto.

5.4.1 Detalhamento da Implementação das Ferramentas

eCrash

Para a instrumentação do código bytecode esta ferramenta utiliza o framework Sofya[20] e para implementação dos algoritmos de busca o *framework* ECJ[21]. O método de otimização utilizado é conhecido como Programação Genética com Tipagem Forte (STGP – *Strongly-Typed Genetic Programming*). Esta abordagem modela a sequência de chamadas de métodos como uma árvore expressando assim as dependências e expõe as sequências viáveis.

O eCrash possui também um módulo de clusterização chamado ATOA que controla as operações entre o instrumentador e o ECJ lendo, se necessário, informações de arquivos para configurar o algoritmo STGP.

Um dos trabalhos desta ferramenta [28] apresenta um estudo empírico com a utilização de um CUT no qual o eCrash obteve bons resultados. A justificativa para a utilização de

[20]http://sofya.unl.edu/
[21]http://cs.gmu.edu/ eclab/projects/ecj/

apenas um problema é que este trabalho era preliminar e futuramente outros experimentos seriam executados.

eToc

Como instrumentador o eToc utiliza uma antiga tecnologia, chamada *OpenJava*[22], que foi descontinuada, a utilização desse *software* acaba por trazer uma grande desvantagem para ferramenta: código intrusivo no código-fonte para permitir instrumentação. Além disto, muitas configurações são necessárias para que o CUT sob teste possa ser lido pela ferramenta o que acaba requisitando tempo do testador. Outro detalhe que limitou a utilização do eToc foi que este não aborda métodos com parâmetros do tipo *array*.

Apesar de todas as limitações mencionadas, a ferramenta está disponível na internet (com código-fonte) e é funcional – dentro de alguns limites. Possui alguma documentação e exemplos. O objetivo é cobrir todos os condicionais do CUT abordado dentro de um tempo limite determinado pelo usuário. Para esta tarefa um GA com operadores genéticos modificados é utilizado. Esta especialização é necessária pois os cromossomos da população do GA são compostos de combinações de chamadas aos métodos do CUT. Portanto o conceito de seleção, recombinação e mutação foi mantido mas a implementação destes foi especializada para poder atacar o problema de geração de dados de teste.

O estudo empírico desta ferramenta utilizou-se de seis CUTs retirados do próprio código-fonte Java SE (pacote *java.util*). Os resultados mostraram que o eToc conseguiu boa cobertura de condicionais em pouco tempo. Falhas intencionalmente inseridas nos CUTs foram, em sua grande maioria, reveladas.

testful

A ferramenta testful utiliza como instrumentador uma modificação do *framework* Soot, o mesmo utilizado no desenvolvimento da ferramenta JET. Apesar disto, esta ferramenta é semi automatizada, ou seja, o passo de instrumentação e o passo de busca devem ser executados separadamente – em dois passos.

O estudo empírico conduzido sobre o testful utilizou 15 CUTs, um dos quais é utilizado neste estudo (*ArrayPartition*), e comparou o desempenho do testful com outras ferramentas. Os resultados mostraram que o testful conseguiu cobrir um maior número de condicionais quando comparado com as outras propostas.

[22] http://www.csg.is.titech.ac.jp/openjava/

5.4.2 Ferramentas Escolhidas

Apenas algumas ferramentas puderam ser comparadas com o JET devido a algumas restrições. É o caso das ferramentas jAutoTest e EvoTest que não disponibilizaram seu código-fonte. Isto impossibilitou o seu estudo e inclusão nesta pesquisa.

Outra restrição é a maneira como cada ferramenta é executada e como se determina o critério de parada. Para o JET, o critério de parada é a cobertura total dos caminhos pré-selecionados – foram escolhidos todos os caminhos independentes para cada CUT, o que resulta na cobertura de todos os nós do CFG de cada CUT. Além disso existe um limite máximo de avaliações da função objetivo que por padrão é de 100.000 mas que pode ser modificado pelo usuário. Já para as ferramentas eToc e testful o critério de parada é a cobertura total de todos os condicionais ou um limite máximo de tempo escolhido. Para o randoop o único critério de parada é o limite máximo de tempo escolhido.

Esta diferença entre o JET e as ferramentas acabou por dificultar a maneira como os dados seriam coletados e comparados. O critério escolhido para avaliar o desempenho foi o tempo gasto para cobrir todos os predicados. Quanto menor este tempo, melhor o desempenho do *software*. Mas para o randoop este critério não pode ser aplicado pois a ferramenta apenas pára de gerar dados após atingir o tempo (em segundos) determinado pelo usuário.

Uma outra dificuldade encontrada para a aplicar a comparação entre as ferramentas é o fato do eCrash ser um *plugin* para o IDE Eclipse. Isto impossibilitou a leitura da cobertura dos condicionais durante a sua execução sobre os CUTs. Mesmo o relatório apresentado por esta ferramenta ao final da execução não detalhava o tempo gasto neste esforço.

Para o testful também houve problemas. Esta ferramenta executa a busca em dois passos distintos sendo que os dois dependem de um ajuste (parâmetro) para determinar o tempo limite máximo. Como explicado na seção 5.1.4, o primeiro momento executa uma busca aleatória, iniciando diversas populações, enquanto que no segundo momento uma pequena porcentagem destas populações (5% 10% melhores adaptadas) são evoluídas utilizando um algoritmo hibrido. A presença destes dois passos, limitados pelo tempo que o usuário determina, impossibilitou a comparação desta ferramenta com o JET. Isto pelo fato de que mesmo o menor valor ajustável (1 segundo) para cada momento (ou seja, 2 segundos no total) é muito maior do que o tempo médio de execução das outras ferramentas – elas cobriam todos os condicionais em poucos milissegundos.

Sendo assim, pelo fato do JET ser um protótipo, e dadas as suas limitações, os CUTs selecionados e o critério de comparação foram escolhidos da melhor maneira para que as ferramentas e o JET pudessem ser comparados. Apesar disto nem todas puderam ser utilizadas. A tabela 5.2 mostra quais ferramentas foram selecionadas dentre as pesquisadas e apresenta uma justificativa resumida do porquê uma determinada ferramenta foi

Nome	Selecionada	Motivo
eCrash	não	Não foi possível obter o tempo de execução por estar inserida como *plugin* no IDE Eclipse
EvoTest	não	Não disponibilizou o código-fonte
eToc	sim	
testful	não	Sua execução ocorre em dois tempos, pré-determinados pelo usuário. Mesmo o tempo mínimo (2 segundos) é muito maior do que o tempo médio de execução das outras ferramentas (milissegundos).
jAutoTest	não	Não disponibilizou o código-fonte
randoop	sim	

Tabela 5.2: Resumo das ferramentas selecionadas

descartada.

5.5 Resultados

Apenas as ferramentas eToc e randoop foram comparadas com o protótipo JET e os resultados são mostrados abaixo. Primeiramente são mostrados gráficos indicando a porcentagem das 200 execuções que obtiveram total cobertura de todos os predicados.

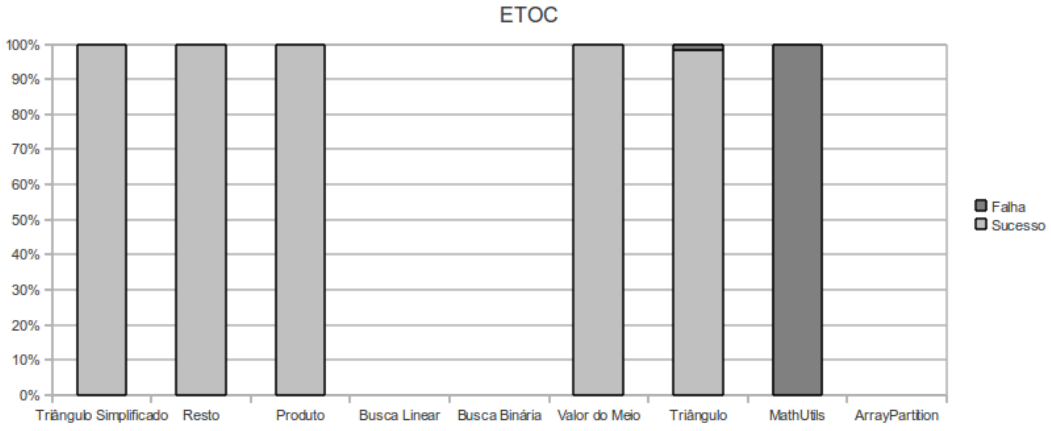


Figura 5.4: eToc - porcentagem de execuções com cobertura total do CUT

É interessante notar que para o eToc os CUTs Busca Linear, Busca Binária e *ArrayPartition* não puderam ser executados pelo fato desta ferramenta não trabalhar com parâmetros do tipo *array*. O JET pode trabalhar estes CUTs pois o pacote *jet-inject* foi utilizado para pré determinar os valores destes parâmetros.

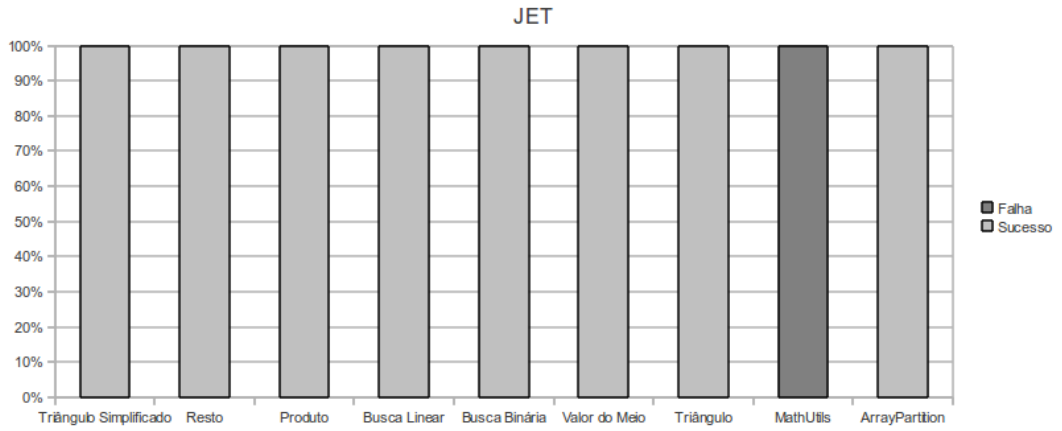


Figura 5.5: JET - porcentagem de execuções com cobertura total do CUT

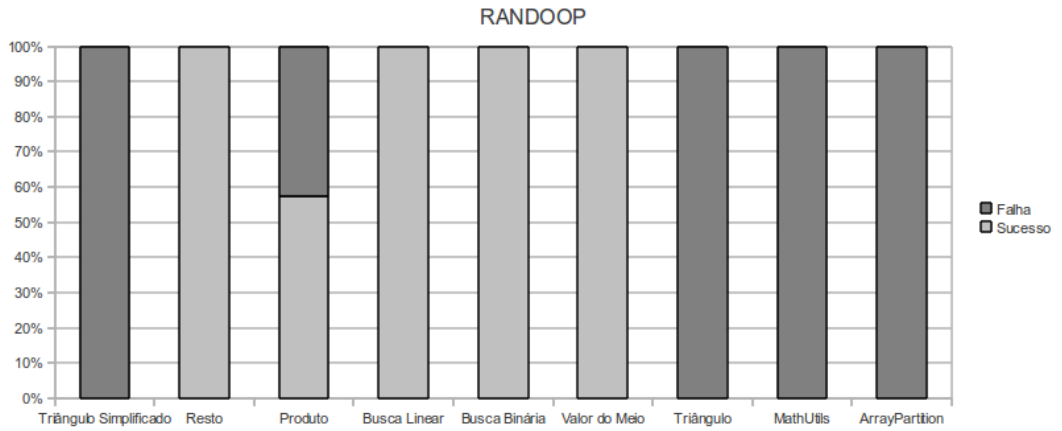


Figura 5.6: randoop - porcentagem de execuções com cobertura total do CUT

Pode-se perceber das figuras 5.4, 5.5 e 5.6 que nenhuma ferramenta conseguiu cobrir completamente, em nenhuma das execuções, o CUT *MathUtils*. Estudando mais a fundo descobriu-se que esta classe possui uma verificação para saber se a soma dos dois parâmetros inteiros é maior (2147483647) ou menor (-2147483648) que o valor Java máximo e mínimo – ou seja, verifica por *overflow*. Nenhuma ferramenta foi capaz de alcançar o código que tratava o *overflow* e por este motivo não cobriu totalmente o CUT em nenhuma das 200 execuções.

Para a ferramenta randoop, os CUTs Triângulo Simplificado, Triângulo e *ArrayPartition* também não foram completamente cobertos em nenhuma das 200 execuções, mesmo com o tempo limite alto (10 segundos). Além disso o CUT Produto teve apenas 58% das

execuções com cobertura completa. Isso se deve ao fato de que existe um condicional que verifica se os dois parâmetros de entrada são iguais a zero e aparentemente a ferramenta não gerou este dado de entrada.

A partir das figuras 5.4, 5.5 e 5.6 pode-se perceber que o protótipo JET foi o que obteve melhor desempenho sobre CUTs. Excetuando-se o CUT *MathUtils* todos foram cobertos em sua totalidade nas 200 execuções.

É importante notar que o JET, que implementa a função objetivo Bueno e Jino e o algoritmo GEO, consegui cobrir o CUT Triângulo, resultado oposto ao mencionado no estudo empírico conduzido na seção 4.5. A razão para o JET conseguir este resultado é o fato deste protótipo executar uma *Thread*[23] para cobrir cada caminho separadamente e cada *Thread* possui o limite de 100.000 avaliações da função objetivo. Já no estudo empírico este limite foi utilizado para cobrir todos os caminhos.

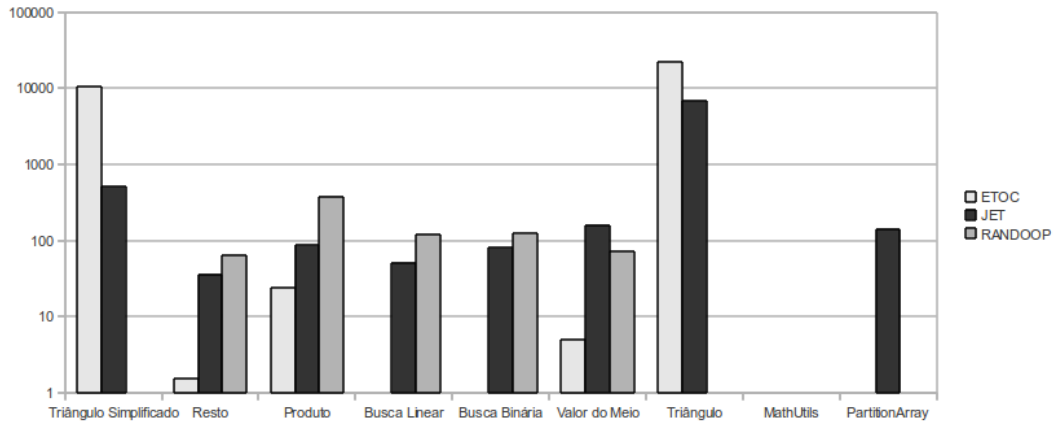


Figura 5.7: Comparação - tempo (milissegundo) médio de execução

Em todas as execuções que conseguiram cobrir totalmente o CUT, o tempo desta tarefa foi contabilizado. A figura 5.7 apresenta o tempo médio de execução para cada CUT/ferramenta.

Como nenhuma ferramenta conseguiu executar com cobertura total o CUT *MathUtils* nenhuma informação foi mostrada. Dos outros oito CUTs restantes percebe-se que o protótipo JET obtém o menor tempo de execução médio em cinco enquanto que a ferramenta eToc consegue em três. Já o randoop, que obteve execuções com cobertura total em cinco CUTs, não teve em nenhuma execução com tempo menor que a dos competidores.

A análise da figura 5.7 mostra que o eToc obteve melhor desempenho que o JET apenas nos CUTs mais "simples". São eles o Resto, Produto e Valor do Meio. Isto pode ser explicado pelo fato do eToc utilizar uma abordagem um pouco diferente de geração

[23]http://download.oracle.com/javase/1.4.2/docs/api/java/lang/Thread.html

de dados do que o JET. Enquanto este último gera dados dentro de um pré determinado domínio o eToc inicialmente gera dados especialmente selecionados antes de gerar dentro de todo o domínio. São dados que geralmente estão nas bordas do domínio, por exemplo: 0, 1, -1, 100, -100... De fato estes três CUTs têm condicionais que são cobertos com estes dados o que fez com que o eToc obtivesse cobertura total com poucas iterações. Já o JET inicia em um ponto aleatório do domínio e somente depois evolui gradativamente para os dados que cobrem esses condicionais.

Portanto pode-se perceber que o protótipo JET obteve bom desempenho em praticamente todos os problemas abordados, principalmente nos mais complexos. Apesar disso sabe-se que diversos fatores podem afetar o desempenho da busca executada pelo algoritmo como a função objetivo implementada, o algoritmo utilizado, sua codificação e até mesmo a codificação interna do CUT. Desta maneira podemos afirmar que o JET tem grande potencial para uma ferramenta caso o protótipo seja evoluído, mas não é possível certificar que o JET é melhor que as outras ferramentas em qualquer problema abordado.

Capítulo 6

Conclusões e Trabalhos Futuros

Este trabalho comparou e avaliou o uso das implementações do algoritmo GEO – em especial o algoritmo GEOreal – na geração dinâmica de dados de teste. Outros algoritmos padrão como a geração aleatória, SGA e RCGA foram implementados para comparação dos resultados. A análise baseou-se no número médio de avaliações da FO para abordar problemas conhecidos na literatura e utilizados por diversos outros trabalhos.

Duas funções objetivo conhecidas foram utilizadas: *Similarity* [13] e a proposta por Bueno e Jino [48]. Ambas tiveram seus domínios desenhados dentro de uma pequena faixa de entrada para avaliação e comparação do resultados em cada SUT. Estes dados auxiliaram na análise dos resultados da execução dos algoritmos sobre os problemas.

Os resultados mostraram que o algoritmo GEOreal tem melhor desempenho quando comparado com os outros algoritmos evolutivos e com a geração aleatória nos SUTs propostos. Quando o problema abordado possuía baixa complexidade a geração aleatória obteve melhor resultado, apesar do GEOreal obter desempenho muito próximo. Os bons resultados deste algoritmo podem ser explicados pelo fato de que algoritmos baseados em codificação real trabalham melhor do que aqueles que usam codificação binária em problemas de alta dimensionalidade [29]. Isso pôde ser observado também pelo RCGA, em relação ao SGA, nos SUTs Triângulo Simplificado e Busca Linear.

Esta conclusão se mostrou verdadeira para a implementação GEOreal independentemente da escolha da função objetivo. Apesar disso ficou claro, através do estudo empírico, que a função objetivo de Bueno e Jino [48] permitiu que os algoritmos obtivessem um melhor desempenho quando comparado com o uso da *Similarity*. Isso pode ser observado através da diminuição do número de avaliações da FO por parte da Bueno e Jino.

Outra grande vantagem na utilização do algoritmo GEOreal na abordagem da geração de dados de teste é o fato deste algoritmo ser simples de ser ajustado através do seu parâmetro τ. O passo de ajuste desta variável para cada SUT foi significantemente menor quando comparado com os algoritmos genéticos – SGA e RCGA – que possuíam

diversas variáveis de ajuste.

Como "prova de conceito" deste estudo o protótipo JET foi desenvolvido implementando o algoritmo GEO e as funções objetivo abordadas neste trabalho. O desempenho deste protótipo foi comparada com diversas ferramentas disponíveis na internet. Os resultados demonstraram que, apesar de algumas limitações do protótipo, existe um grande potencial para uma futura ferramenta. Os dados coletados do estudo empírico apontam que o JET tem bom desempenho ao abordar o problema de geração de dados de teste e este superou as ferramentas comparadas ao se considerar o tempo de execução.

Este trabalho contribuiu à literatura da área publicando um artigo com os primeiros resultados deste estudo no *Workshop* SAST - *4th Brazilian Workshop on Systematic and Automated Software Testing*. Eis a referência:

- Buzzo A. V., Martins E., Sousa F. L. (2010), Uso do algoritmo GEOreal para abordar a geração automática de dados de teste. In: *4th Brazilian Workshop on Systematic and Automated Software Testing*, 11 de Novembro, 2010, Natal, Brazil.

6.1 Trabalhos Futuros

Como proposta de trabalho futuro mais algumas funções objetivos poderia ser pesquisadas e codificadas explorando assim outros critérios de teste. Também seria desejável adaptar o GEO para que ele trabalhasse com outros tipos de algoritmos, formando assim um algoritmo híbrido. Outra proposta seria incluir outras funções objetivo na avaliação da geração de dados de teste utilizando o conceito multi-objetivo.

6.2 Trabalhos Futuros para o JET

O JET pode ser transformado em uma ferramenta de geração de testes competitiva com as presentes no mercado se alguns itens forem implementados. Inicialmente o mais importante é resolver as limitações mencionadas na seção 5.2.3:

1. Abordar classes e não somente à métodos.
2. Tratamento *stateful* para a instância da classe durante as iterações.
3. Abordar métodos com parâmetros de todos os tipos Java (*string*, *boolean*, etc).
4. Mudar o critério de parada para tempo e não número de iterações da FO.
5. Implementar uma GUI. Talvez como *plugin* do IDE Eclipse.

Bibliografia

[1] Baresel A., Pohlheim H., and Sadeghipour S. Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms. In *GECCO 2003: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 2428–2441. IEEE Computer Society Press, 2003.

[2] Ould M. A. Testing a challenge to method and tool developers. *Software Engineering Journal*, pages 59–64, March 1991.

[3] Watkins A. and Hufnagel E. M. Evolutionary test data generation: a comparison of fitness functions. *Software Practice and Experience*, 36(1):95–116, 2006.

[4] Windisch A., Wappler S., and Wegener J. Applying particle swarm optimization to software testing. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1121–1128, New York, NY, USA, 2007. ACM.

[5] Fogel D. B. An introduction to simulated evolutionary computation. *IEEE Transactions on Neural Networks*, 5(1):3–14, 1994.

[6] Korel B. Automated software test data generation. *IEEE Trans. Software Eng*, 16(Issue 8):870–879, 1990.

[7] Korel B. Automated test data generation for programs with procedures. *ACM Transactions on Software Engineering and Methodology*, 5(1):209–215, January 1996.

[8] Meyer B., Ciupa I., Leitner A., and Liu L. Automatic testing of objectoriented software. In *Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science*, pages 114–129. Springer, 2007.

[9] Luciano Baresi, Matteo Miraz, and Pierluigi Plebani. A flexible and semantic-aware publication infrastructure for web services. In *Advanced Information Systems Engineering*, volume 5074, pages 435–449. Springer Berlin / Heidelberg, 2008.

[10] Bourhfir C., Dssouli R., and Aboulhamid E. M. Automatic test generation fo efsm-based systems. Relatório técnico iro 1043, University of Montreal, Canada, August 1996.

[11] Ince D. C. The automatic generation of test data. *The Computer Journal*, 30(Issue 1):63–68, 1987.

[12] King J. C. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[13] Lin J. C. and Yeh P. L. Automatic test data generation for path testing using gas. *Information Sciences*, 131(1-4):47–64, 2001.

[14] Michael C. C., MacGraw G., and Schatz M. A. Generating software test data by evolution. *IEEE Transaction on Software Engineering*, 27(Issue 12):1085–1110, December 2001.

[15] Pacheco C. and Ernst M. D. Randoop: Feedback-directed random testing for java. In *OOPSLA 2007 Companion*, Montreal, Canada, October 2007.

[16] Pacheco C., Lahiri S. K., and Ball T. Finding errors in .net with feedback-directed random testing. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, pages 87–96, 2008.

[17] Alba E. and Chicano J. F. Software testing with evolutionary strategies. *Proc. of Workshop on Rapid Integration of Software Engineering Techniques*, October 2005.

[18] Eiben A. E. and Smith J. E. *Introduction to Evolutionary Computing*. Springer, 2003.

[19] Goldberg D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addilson-Wesley, 1989.

[20] Howden W. E. Reliability of the path analysis testing strategy. *IEEE Transactions of Software Engineering*, SE02(3), September 1976.

[21] Myers G. *The Art of Software Testing*. Wiley, 1979.

[22] Holland J. H. *Adaptation in natural and articial systems*. University of Michigan Press, 1975.

[23] Sthamer H. *The Automatic Generation of Software Test Data Using Genetic Algorithms*. PhD thesis, University of Glamorgan, Pontyprid, Wales, Great Britain, 1996.

[24] Wolpert D. H. and Macready W. G. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, The Santa Fé Institute, Santa Fé, 1995.

[25] Sommerville I. *Engenharia de Software*. Pearson Education, 6 edition, 2003.

[26] Harrold M. J. Testing: a roadmap. In *ICSE – Future os SE Track*, pages 61–72, 2000.

[27] McCabe T. J. A complexity measure. In *ICSE'76: Proc. of the 2nd international conference on Software engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[28] Ribeiro J., Zenha-Rela M. A., and Fernadéz de Vega F. ecrash: a framework for performing evolutionary testing on third-party java components. In *Proceedings of the I Jornadas sobre Algoritmos Evolutivos y Mataheuristicas (JAEM)*, pages 137–144, Zaragoza, 2007.

[29] Von Zuben F. J. Computação evolutiva: Uma abordagem pragmática. In *I Jornada de Estudos em Computação de Piracicaba e Região (1a JECOMP)*, volume 1, pages 25–45. Anais da I Jornada de Estudos em Computação de Piracicaba e Região, 2000.

[30] Wegener J., Baresel A., and Sthamer H. Evolucionay test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[31] Lakhotia K., Harman M., and McMinn P. A multi-objective approach to search-based test data generation. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1098–1105, London, England, 2007. ACM.

[32] Baresi L., Lanzi P. L., and Miraz M. Testful: An evolutionary test approach for java. In *Software Testing, Verification and Validation (ICST)*, pages 185–194, 2009.

[33] Baresi L. and Miraz M. Testful: automatic unit-test generation for java classes. In *ICSE '10 Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, volume 2, 2010.

[34] Chambers L. *The Practical Handbook of Genetic Algorithms: Applications*. Chapman & Hall, 2001.

[35] Lopes I. M. L. Controle de atitude de satélites rígido-flexíveis usando a otimização extrema generalizada com abordagem multi-objetivo. Mestrado em mecânica espacial e controle, Instituto Nacional de Pesquisas Espaciais, 2008.

[36] Sousa F. L. *Otimização Extrema Generalizada: Um novo algoritmo estocástico para o projeto ótimo*. PhD thesis, INPE, São José dos Campos, SP, Brasil, 2002.

[37] Sousa F. L., Ramos F. M., Paglione P., and Giardi R. M. New stochastic algorithm for design optimization. *AIAA Journal*, 41(9):1808–1818, September 2003.

[38] Sousa F. L., Vlassov V., and Ramos F. M. Generalized extremal optimization: An application in heat pipe design. *Applied Mathematical Modeling*, 28:911–931, 2004.

[39] Harman M. Automated test data generation using search based software engineering. *Automation of Software Test*, 2:20–26, 2007.

[40] Harman M. The current state and future of search based software engineering. In *Future of Software Engineering 2007*. IEEE Computer Society, 2007.

[41] Harman M. and McMinn P. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *International Symposium on Software Testing and Analysis*, 2007.

[42] Miraz M., Lanzi P. L., and Baresi L. Improving evolutionary testing by means of efficiency enhancement techniques. In *Proceedings of the IEEE Congress on Evolutionary Computation*, 2010.

[43] Mansour N. and Salame M. Data generation for path testing. *Software Quality Journal*, 12(2):121–136, 2004.

[44] Tran N. and Deville Yves. Automatic test data generation for programs with integer and float variables. In *16th IEEE International Conference on Automated Software Engineering*, pages 3–21, 2001.

[45] Boaventura Netto P. O. *Grafos: Teoria, Modelos e Algoritmos*. Edgar Blucher, São Paulo, 1996.

[46] Infotech State of the Art Report. *Software Testing Volume 1: Analysis and Bibliography*. Infotech International, 1979.

[47] Bak P. and Sneppen K. Punctuated equilibrium and criticality in a simple model of evolution. *Physical Review Letters*, 71(24):4083–4086, December 1993.

[48] Bueno P. and Jino M. Automatic test data generation for program paths using genetic algorithms. *International Journal of Software Engineering and Knowledge Engineering*, 12(6):691–709, 2002.

[49] McMinn P. Search-based software test data generation: a survey. *Software Testing, Verification and reliability*, 14(2):105–156, 2004.

[50] Pargas R. P., Harrold M. J., and Peck R. P. Test-data generation using genetic algorithms. *Software Testing, Verification & Reliability*, 9(4):263–282, 1999.

[51] Tonella P. Evolutionary testing of classes. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128, 2004.

[52] Guo Q., Hierons R. M., Harman M., and Derderian K. Constructing multiple unique input/output sequences using metaheuristic optimisation techniques. *IEE Proceedings — Software*, 152(3):127–140, 2005.

[53] Santos J. P. P. R. Niche search: an evolutionary algorithm for global optimization, parallel problem solving from nature iv. *Lecture Notes in Computer Science*, 1411:430–440, 1996.

[54] Boettcher S. and Percus A. G. Optimization with extremal dynamics. *Physical Review Letters*, 86:5211–5214, 2001.

[55] Luke S. Ecj 16: A java evolutionary computation library, 2007.

[56] Pressman R. S. *Engenharia de Software*. McGraw-Hill, 6 edition, 2006.

[57] Yoo S. and Harman M. Pareto efficient multi-objective test case selection. In *International Symposium on Software Testing and Analysis*, 2007.

[58] Abreu B. T. Uma abordagem evolutiva para a geração automática de dados de teste. Master's thesis, IC/Unicamp, Campinas/SP, 2006.

[59] Abreu B. T., Martins E., and Sousa F. L. Automatic test data generation for path testing using a new stochastic algorithm. In *XIX Simpósio Brasileiro de Engenharia de Software*, volume 19, pages 247–262, Uberlândia, Brasil, 2005.

[60] Abreu B. T., Martins E., and Sousa F. L. Generalized extermal optimization: Anattractive alternative for test data generation. In *Proceedings of the 2007 Genetic and Evolutionary Computation Conference*, GECCO '07, page 1138. ACM, 2007.

[61] Back T. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, 1996.

[62] Binder R. V. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 1999.

[63] Joachim Wegener and Oliver Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *Proceedings of GECCO*, pages 1400–1412. Springer-Verlag, 2004.

[64] T. Yano, E. Martins, and F. L. De Sousa. Generating feasible test paths from an executable model using a multi-objective approach. In *Proc. SBST'10: 3rd Int. Workshop on Search-Based Software Testing*, pages 236–239, Paris, France, 2010.

[65] Michalewicz Z. and Schoenauer M. Evolutionary algorithms for constrained parameter optimization problems. *Evolutionary Computation*, 4(1):1–32, 1996.

[66] Michalewicz Z., Hinterding R., and Michalewicz M. Evolutionary algorithms. In Fuzzy Evolutionary Computation, editor, *Witold Pedrycz*, pages 3–31, Kluwer Academic, Boston, 1997.

Apêndice A

Análise dos SUTs Restantes do Estudo Empírico

A.1 Resto

Dentre todos os SUTs estudados o SUT Resto é o problema mais simples. Como pode ser observado na tabela 4.1 ele possui a menor complexidade ciclomática dentre todos os SUTs. Duas são as variáveis de entrada sendo que a segunda é repetidamente subtraída da primeira até que o valor se torne menor que a segunda variável.

```
0 private int rest(final int x, final int y) {
1     int quotient = 0;
1     int rest = x;
2     while (rest >= y & y > 0) {
3           rest -= y;
3           quotient++;
      }
4     return rest;
 }
```

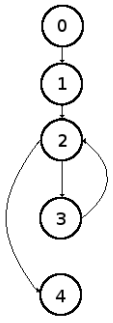


Figura A.1: CFG - Resto

A figura A.1 mostra o GFC deste problema e a partir dele foram escolhidos seis

caminhos. Neste SUT, diferentemente do problema triângulo simplificado, temos um *loop* interno. Dentre os caminhos escolhidos para serem cobertos o primeiro não executa o *loop* nenhuma vez (0-1-2-4) e outros cinco devem executar o *loop* uma vez a mais que o caminho anterior (0-1-**2-3**-2-4, 0-1-**2-3-2-3**-2-4, 0-1-**2-3-2-3-2-3**-2-4 ...).

A.1.1 Cobertura Média dos Algoritmos

O número de avaliações máximas para este SUT foi limitada a 100.000. Utilizando a FO *Similarity* (figura A.2) percebe-se que rapidamente os algoritmos GEOcan, GEOvar e GEOreal atingem a cobertura total com menos de 4.000 avaliações. Já os outros algoritmos apenas atingem esta marca com mais de 75.000 avaliações. É possível observar os algoritmos SGA e RCGA são, em média, os mais demorados para executar este SUT enquanto que a geração aleatória e GEOreal são os mais rápidos. Quando a FO Bueno e Jino é utilizada percebe-se que o GEOcan, GEOvar e GEOreal conseguem obter a cobertura total em metade do número de avaliações (figura A.3).

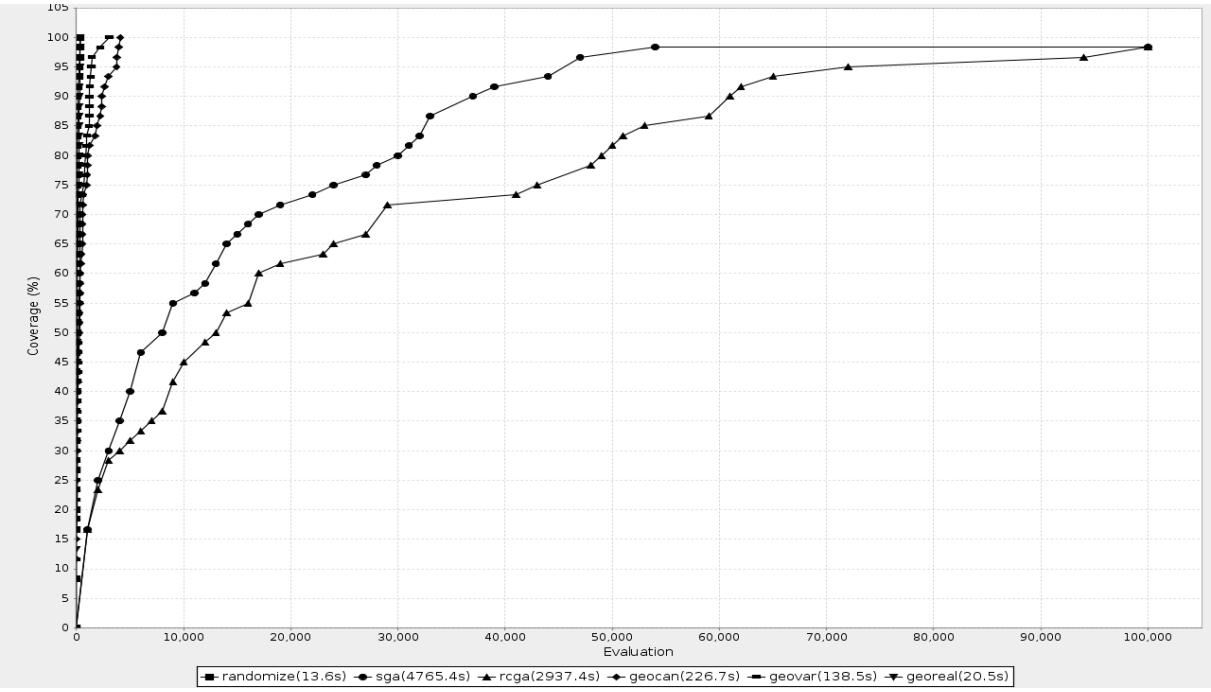


Figura A.2: SUT Resto (20.000 avaliações) – cobertura média dos algoritmos usando FO *Similarity*

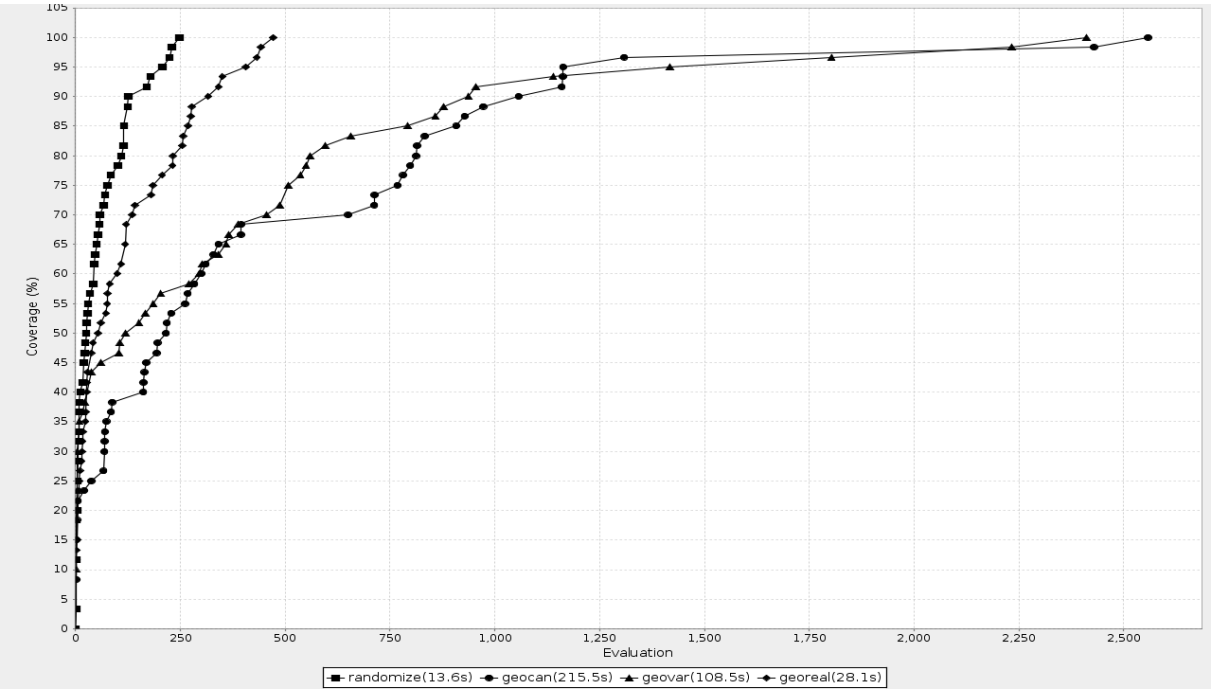


Figura A.3: SUT Resto (2.600 avaliações) – cobertura média dos algoritmos usando FO Bueno e Jino

A.1.2 Avaliação da Função Objetivo *Similarity*

A figura A.4 aponta claramente dois planos bem distintos. No mais baixo temos metade dos pontos do domínio e todos os que possuem o valor de Y = 0. Este plano mostra todos os pares (X,Y) onde X é menor que Y, ou seja, a primeira variável de entrada é menor que a segunda. Esses pares conduzem o SUTs pelo primeiro caminho da lista que é o que não executa o *loop* nenhuma vez. Os pontos que possuem Y = 0 também não executam o *loop* nenhuma vez pois é umas da condições do SUT e do condicional do *loop*: o divisor (segunda variável de entrada) não deve ser zero.

O segundo plano é formado pelos pontos restantes do domínio. São todos os pontos que possuem a primeira variável de entrada maior que a segunda e que executam o *loop* interno do SUT uma ou mais vezes. Percebe-se aqui, pela primeira vez, uma característica interessante da *Similarity*: os pares (X,Y) que executaram o *loop* uma ou mais vezes não foram avaliados com valores diferentes. Todos os pontos (X,Y) que executam os caminhos 0-1-(2-3)-2-4 (figura A.1) são avaliados com o mesmo valor. Isso indica uma incapacidade da FO de trabalhar com *loops*: caminhos que executam o *loop* mais ou menos vezes são avaliados com o mesmo valor. Isto afeta diretamente um algoritmo que utiliza esta FO em convergir para a solução (caminho) procurada(o).

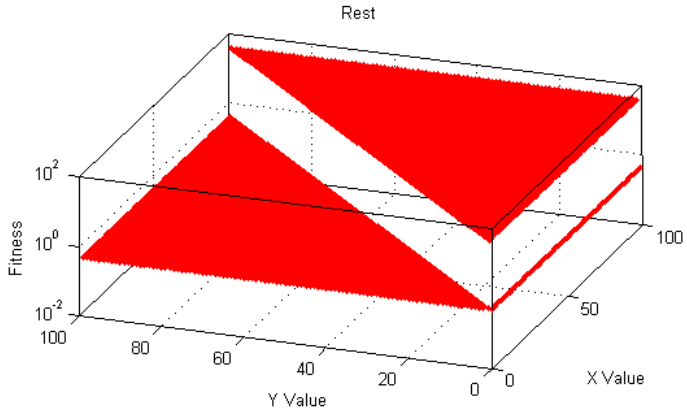


Figura A.4: Avaliação da FO *Similarity* - Resto

A.1.3 Avaliação da Função Objetivo Bueno e Jino

As figuras A.5 e A.6 mostram um comportamento bem diferente do apresentado pela avaliação do domínio da *Similarity* para este problema. Pode-se ver diversos planos inclinados (em formato de triângulo). Como esta FO é de maximização vemos que o pequeno plano que possui os pontos com maior valor de avaliação são os que conduzem o SUT pelo caminho (0-1-2-3-2-3-2-3-2-3-2-3-2-4) escolhido como alvo. Este caminho executa o *loop* interno do SUT 5 vezes.

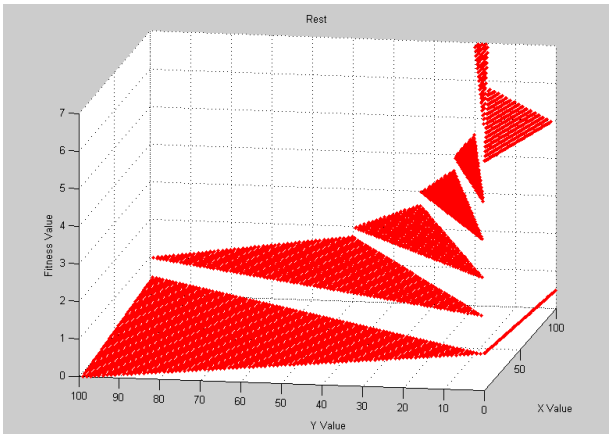


Figura A.5: Avaliação da FO Bueno e Jino - Resto

Os planos inferiores seguintes são os conjuntos de pares (X,Y) que executam o *loop* do SUT menos vezes. Temos o plano de pontos que executa o *loop* quatro vezes, três, etc. até uma vez. A partir dai temos o plano que é composta de todos os pontos (X,Y) onde Y é maior que X ou onde Y = 0 (pequena reta na lateral do gráfico). Este plano mostra

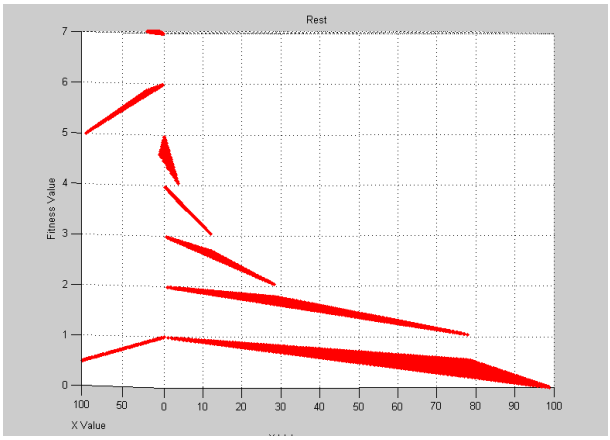


Figura A.6: Avaliação da FO Bueno e Jino - Resto

os pontos que não executam o *loop* nenhuma vez. Veja que quão mais distantes do plano com os pontos (X,Y) que executam o caminho alvo, menor a avaliação da FO.

Finamente temos um pequeno plano visto do lado direito da figura A.5. Esta curva compõe os pontos (X,Y) que executam o *loop* interno do SUT mais de cinco vezes. Também podemos perceber que os pontos que executam o *loop* mais vezes que os outros tem sua avaliação cada vez com valor mais baixo (perceptível na figura A.6 do lado esquerdo).

A.2 Produto

O SUT Produto é muito parecido com o SUT Resto. Apesar de sua complexidade ciclomática ser um pouco mais alta que a do Resto seu código interno é semelhante por possuir um *loop* interno que soma repetidamente a primeira variável de entrada o número de vezes indicada pela segunda variável de entrada. Desta maneira o resultado retornado representa a multiplicação entre a primeira e segunda variáveis de entrada.

A partir do GFC deste SUT A.7 seis caminhos alvos foram escolhidos. O primeiro passa pela condicional que é executada quando uma das variáveis é igual a zero (0-1-2-3-7). O segundo caminho é o fluxo para quando a segunda variável de entrada vale um e desta maneira o *loop* interno não é executado nenhuma vez (0-1-2-4-5-7). Por fim, os outros quatro caminhos são execuções incrementadas de um em relação ao caminho anterior (0-1-2-4-**5-6**-5-7, 0-1-2-4-**5-6-5-6**-5-7 ...).

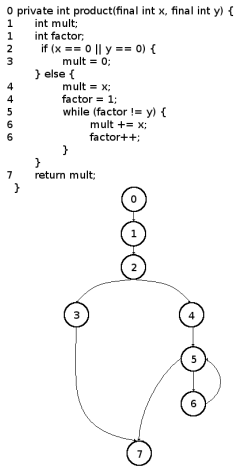


Figura A.7: CFG - Produto

Cobertura Média dos Algoritmos

Neste SUT o número de avaliações da FO foi limitado a 100.000. Observando A.8, percebe-se que apenas os algoritmos GEOcan, GEOvar, GEOreal e a geração aleatória são capazes de alcançar 100% de cobertura do problema. O algoritmo SGA alcança próximo a 95% de cobertura média enquanto que o RCGA atinge no máximo 80%.

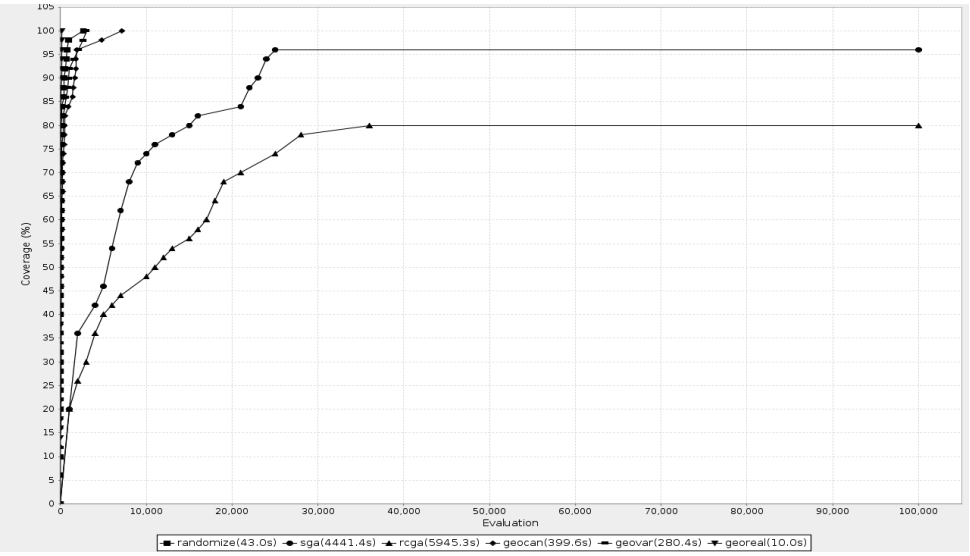


Figura A.8: SUT Produto (100.000 avaliações) – cobertura média dos algoritmos usando FO *Similarity*

Aqui percebe-se uma característica interessante no RCGA que é um algoritmo base-

ado em uma metaheurística utilizando genes reais: quando há um condicional no SUT que exige que um dado de entrada seja exatamente zero o algoritmo dificilmente consegue gerar este dado. Estes algoritmos trabalham com dados de entrada utilizando números reais e estes possuem diversas casas decimais (cerca de cinco). Os algoritmos dificilmente conseguirão gerar o valor exatamento zero. Isto explica porque o RCGA não conseguiram cobrir o quinto e último caminho do SUT permanecendo com cobertura média de 80%. Este último caminho exigia que um dos dois dados de entrada fosse zero.

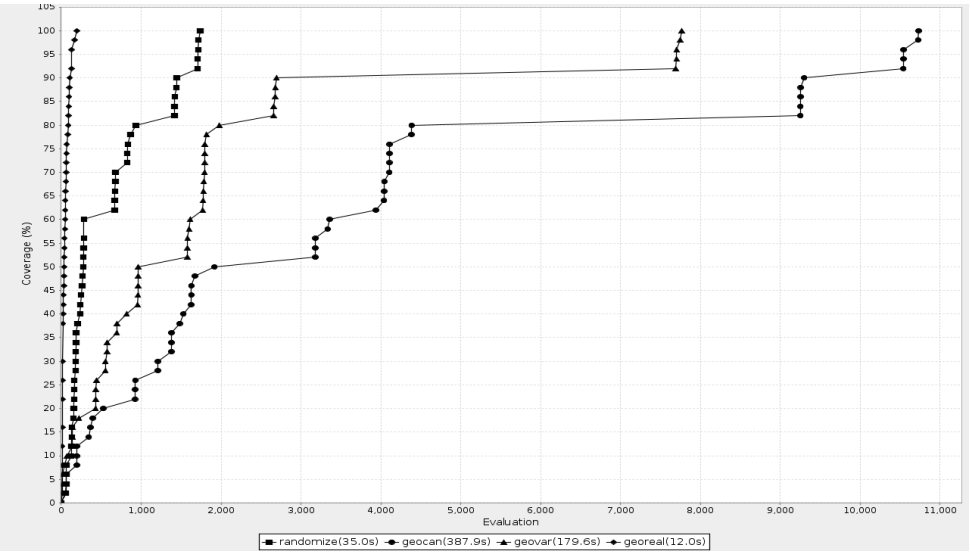


Figura A.9: SUT Produt (11.000 avaliações) – cobertura média dos algoritmos usando FO Bueno e Jino

A partir da figura A.9 observamos as mesmas características apontadas para a *Similarity*: os algoritmos GEOcan, GEOvar, GEOreal e a geração aleatória alcançam 100% de cobertura.

A.2.1 Avaliação da Função Objetivo *Similarity*

O gráfico A.10 é claro em mostrar três planos distintos. O primeiro referente aos pares (X,Y) que executam o primeiro caminho: onde X ou Y valem zero. O segundo plano é composto pelos pontos onde Y vale um. Todos os outros pares de pontos executam o *loop* interno do SUT pelo menos um vez. É possível notar que todos eles foram avaliados com o mesmo valor pela função objetivo independente do número de execuções do *loop*. Novamente temos um indicativo de que a FO *Similarity* é incapaz de avaliar com valores diferentes caminhos que executam um ciclo (*loop*) uma ou mais vezes. Isso acaba por não

guiar corretamente um algoritmo baseado nesta FO a encontrar o caminho alvo.

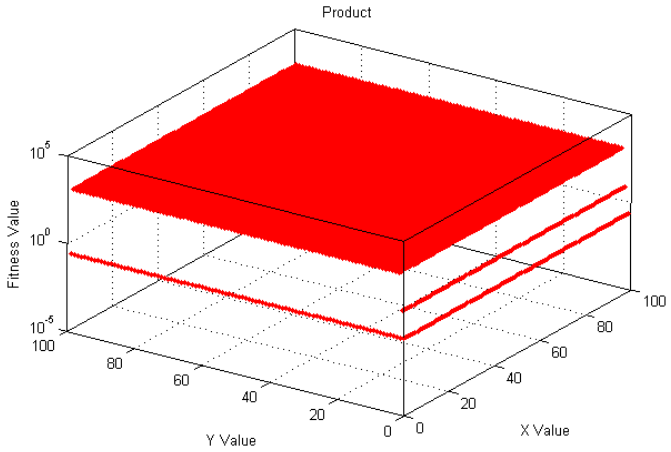


Figura A.10: Avaliação da FO *Similarity* - Produto

A.2.2 Avaliação da Função Objetivo Bueno e Jino

Podemos observar nas figuras A.11 e A.12 um comportamento parecido com o das figuras A.5 e A.6 (SUT resto). Existem diversos conjuntos de pontos (X,Y) formando cerca de cinco planos triangulares inclinados. Vemos também uma pequena quantidade de pontos referentes àqueles com (0,Y) ou (X,0). Este último conjunto de pontos representa o quinto e último caminho do SUT Produto que não foi coberto pelos algoritmos que trabalham com números reais (RCGA e GEOreal).

Dentre os planos triangulares temos um que é o mais alto. Este é o conjunto de pontos que foram bem avaliados pela FO pois são os que exercitam o caminho alvo: 0-1-2-4-5-6-5-6-5-6-5-7 (veja figura A.7). Este caminho exercita o *loop* interno do SUT três vezes. Em seguida temos um plano mais baixo (do lado direito na figura A.11). Este é o conjunto de pontos que exercitam o *loop* duas vezes. O próximo plano segue a mesma ideia, para a execução do *loop* interno apenas uma vez. Finalmente temos o maior plano de pontos que não exercitam o *loop* nenhuma vez. Não podemos esquecer do último plano mais a esquerda na figura A.11 formado pelos pontos que exercitam o *loop* do SUT mais de três vezes. Conforme os pares (X,Y) exercitam o *loop* mais vezes, mais estes são penalizados pela FO tendo sua avaliação cada vez menor. Por este motivo o plano é inclinado como verificado no canto esquerdo da figura A.12.

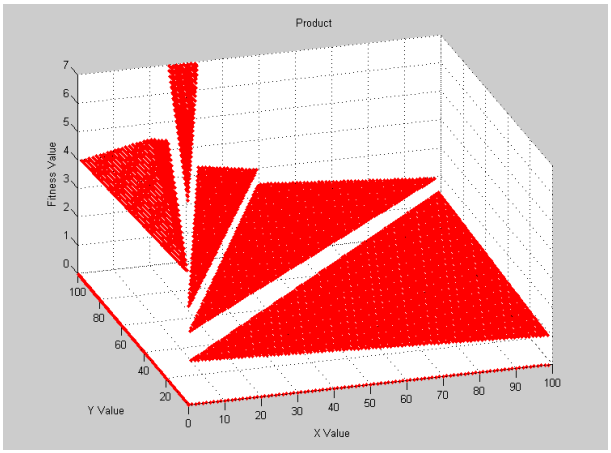


Figura A.11: Avaliação da FO Bueno e Jino - Produto

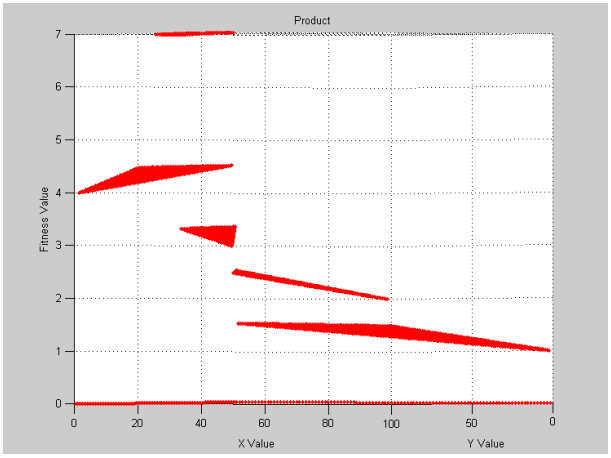


Figura A.12: Avaliação da FO Bueno e Jino - Produto

A.3 Busca Linear

O SUT Busca Linear é um dentre os dois SUTs que possuem apenas uma variável de entrada. Na busca linear um vetor é inicializado aleatoriamente com treze valores dentro do domínio. O vetor foi inicializado com treze valores pois este é o mesmo número utilizado por Abreu [58]. Esta inicialização é feita apenas uma vez antes das execuções do SUTs na busca pela cobertura dos caminhos. Todas as execuções utilizam o mesmo vetor de busca.

Este SUT também é muito parecido com o SUT Resto e o SUT Produto pois seu código interno é composto por um único *loop*. O objetivo deste laço é avançar gradualmente desde o primeiro valor do vetor até o último verificando se o valor da variável de entrada está na posição atual do vetor. Caso o valor seja encontrado a iteração com o

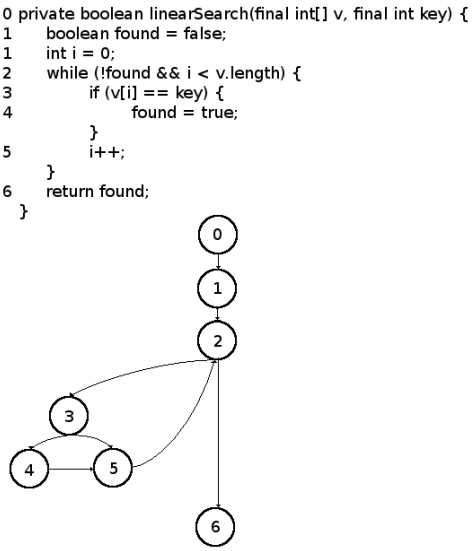


Figura A.13: CFG - Busca Linear

laço é parada e o método retorna o booleano *true*. Caso o valor não seja encontrado em nenhuma iteração o laço é terminado com o fim do vetor e o booleano *false* é retornado.

Cinco são os caminhos que foram escolhidos para serem cobertos a partir da figura A.13. O primeiro é o único caminho que não executa nenhuma vez o laço (0-1-2-6) pois é o caminho que encontra o valor da variável de entrada logo na primeira posição do vetor. Os outros quatro caminhos executam uma vez a mais o *loop* que o caminho anterior (0-1-**2-3-4-5-2**-6, 0-1-**2-3-5-2-3-4-5-2**-6, ...).

A.3.1 Cobertura Média dos Algoritmos

O número de avaliações da função objetivo foi limitada a 400.000. Em A.14 é possível observar que praticamente todos os algoritmos, excetuando-se o SGA, RCGA e GEOreal, conseguem atingir 100% de cobertura. A geração aleatória é mais eficiente em conseguir esta marca com aproximadamente 175.000 avaliações. Já o algoritmo GEOreal não conseguiu atingir mais que 90% de cobertura e estagnou nela a partir da 10.000 avaliação. O algoritmo SGA não conseguiu cobrir qualquer caminho e permaneceu com 0% de cobertura.

Apesar dos algoritmos GEOreal e RCGA apresentarem uma curva que não atinge o valor de 100% isto não significa que todas as execuções não atingiram esta marca. Significa que pelo menos uma execução não conseguiu cobertura de 100%. De fato, ao verificar a análise do número de avaliações da FO (tabela 4.3), percebe-se que a média de avaliações

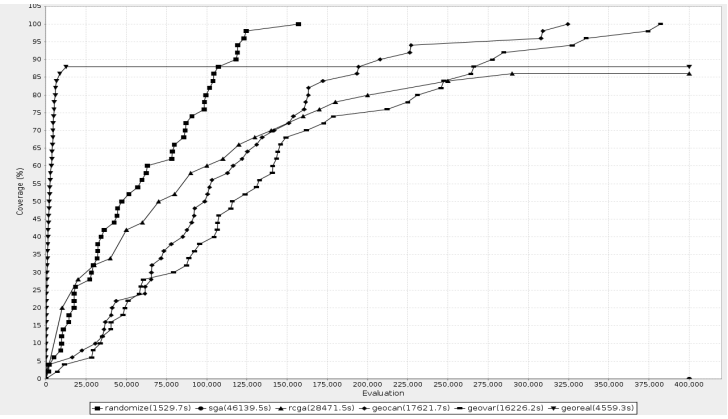


Figura A.14: SUT Busca Linear (400.000 avaliações) – cobertura média dos algoritmos usando FO *Similarity*

é de 6.305 para a *Similarity*. O mesmo se aplica para o RCGA.

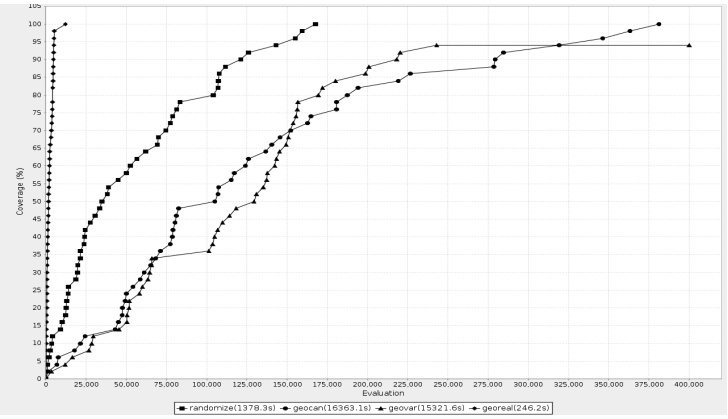


Figura A.15: SUT Busca Linear (400.000 avaliações) – cobertura média dos algoritmos usando FO Bueno e Jino

O comportamento dos algoritmos GEOcan, GEOvar e GEOreal tiveram pouca mudança quando comparada a cobertura da Bueno e Jino A.15 com *Similarity*.

A.3.2 Avaliação da Função Objetivo *Similarity*

A figura A.16 mostra a avaliação da FO deste SUT plotado no domínio total da única variável de entrada [0,16838]. Por este motivo obtemos um gráfico bidimensional, diferente dos anteriores. No eixo X temos os valores da variável de entrada no domínio do SUT. O eixo Y mostra os valores da avaliação da função objetivo em escala logarítmica

para melhor visualização dos dados.

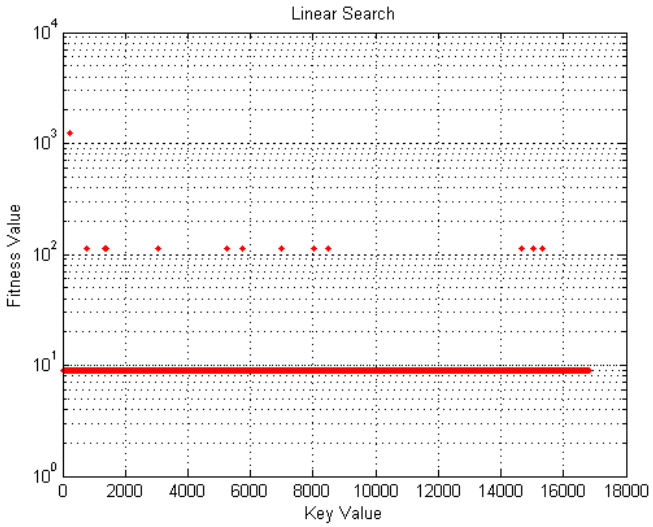


Figura A.16: Avaliação da FO *Similarity* - Busca Linear

Existem três níveis visíveis onde todos os pontos do domínio são apresentados. No nível mais alto temos um único ponto. Este é o valor da variável de entrada que estava na primeira posição do vetor e consequentemente é o ponto que representa a avaliação da FO para o primeiro caminho. Todos os outros doze pontos (existem dois pontos muito próximos em $x \sim 1800$) que estão no segundo nível mostram que a FO avalia com o mesmo valor todos os caminhos que passam pelo *loop* do SUT, independentemente de quantas vezes ele é executado. Isso significa que os outros quatro caminhos do SUT têm o mesmo valor de avaliação pela FO. Por fim, o último nível (o mais baixo) coloca todos os pontos que não estão no vetor com um mesmo valor de *fitness* pois nestes valores de entrada o *loop* interno do SUT será executado a mesma quantidade de vezes até o término da condição do laço.

Mais uma vez podemos notar a incapacidade da *Similarity* em avaliar com valores diferentes caminhos que executam um *loop* um número diferente de iterações. Como este SUT possui quatro caminhos que se diferem no número de vezes em que o *loop* é executado, qualquer algoritmo que possua uma metaheurística que utilize esta FO não conseguirá distinguir a diferença entre os caminhos.

A.3.3 Avaliação da Função Objetivo Bueno e Jino

No gráfico de avaliação do domínio da FO Bueno e Jino (figura A.17 observa-se claramente cinco pontos distintos e uma curva com valor Y máximo (valor 10) em $X \sim 4700$. Os cinco pontos representam os valores de X que executam o problema e exercitam os cinco caminhos escolhidos para este SUT. O ponto mais alto (Y = 12) representa último caminho desta lista sendo o caminho alvo escolhido na avaliação da FO.

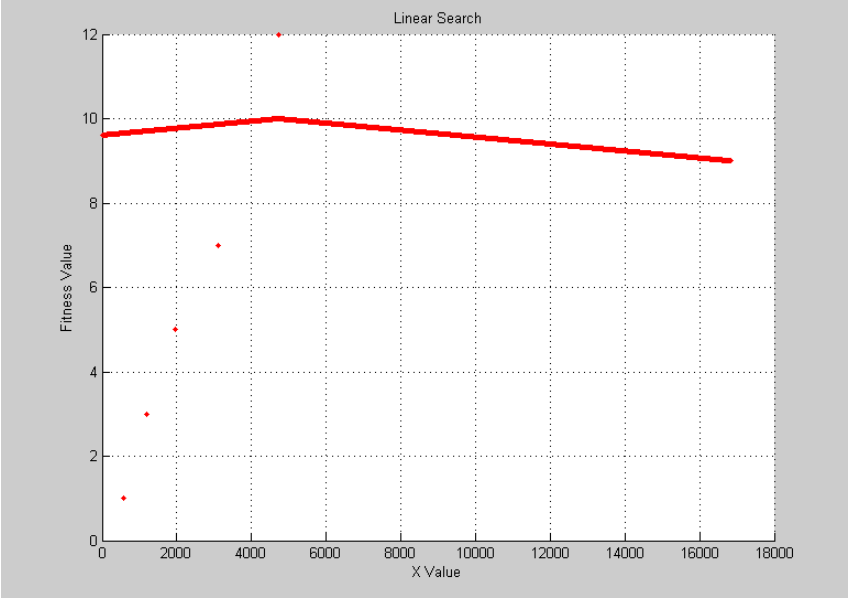


Figura A.17: Avaliação da FO Bueno e Jino - Busca Linear

Todos os outros pontos entre 0 e 16838 (domínio do problema – veja tabela 4.1) exercitam o caminho que executa o *loop* interno do SUT até que o vetor onde existem os valores buscados seja totalmente percorrido. Este caminho desvia-se do caminho alvo escolhido e portanto tem seu valor avaliado pela FO penalizado por um fator. Este fator de penalização vai diminuindo quanto mais próximo X for do valor que conduz à execução do SUT pelo caminho alvo (em $X \sim 4700$). Isto explica o formato da curva com pico em $X \sim 4700$.

A.4 Busca Binária

O SUT Busca Binária é de longe o problema que possui maior número de caminhos que foram escolhidos para serem cobertos: são doze ao todo. Eles foram obtidos a partir da figura A.18.

Além do SUT Busca Linear este SUT é o outro que possui apenas uma variável

```
0 private boolean binarySearch(final int[] v, final float key) {
1     boolean found = false;
1     int low = 0;
1     int high = v.length - 1;
1     int mid;
2     while (low <= high && !found) {
3           mid = (low + high) / 2;
4           if (key < v[mid]) {
5                 high = mid - 1;
            } else if (key > v[mid]) {
6                 low = mid + 1;
            } else {
7                 found = true;
            }
      }
8     return found;
 }
```

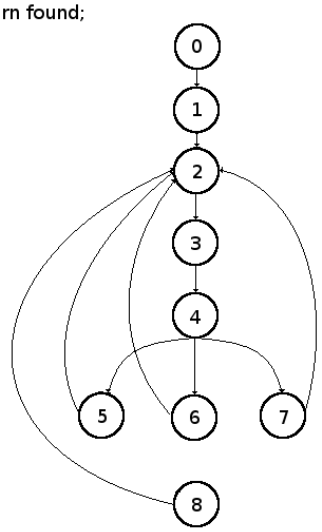


Figura A.18: CFG - Busca Binária

de entrada. Antes de qualquer execução, um vetor com quarenta valores aleatoriamente escolhidos é inicializado (mesmo número utilizado por Abreu [58]. Este vetor é ordenado e é o mesmo utilizado em todas as execuções de cada algoritmo.

Este SUT foi implementado usando a forma iterativa do algoritmo busca binária. Existem implementações que utilizam a forma recursiva. Dentro do código deste SUT, temos um *loop* com um condicional em seu interior. Este *loop* apenas termina quando o valor da variável de entrada é encontrado em alguma posição do vetor ou quando este não é encontrado pois não está no vetor. O condicional no interior do *loop* é responsável por mudar a localização do índice de busca do algoritmo o colocando na metade superior ou inferior do vetor conforme a comparação feita entre a chave (variável de entrada) e o valor na posição do índice de busca atual.

Todos os doze caminhos escolhidos exigem mais de uma iteração do *loop* do SUT. O primeiro caminho (0-1-2-3-**4-7**-4-8) é aquele encontra a chave (variável de entrada) no

índice dez do vetor, ou seja duas iterações do *loop*: inicialmente o índice esta no meio do vetor (índice vinte) e em seguida na metade inferior (índice dez). O caminho seguinte (0-1-2-3-**4-6-4-7**-4-8) encontra a chave no índice trinta (duas iterações – na segunda iteração o índice é deslocado para a metade superior do vetor). Os caminhos seguem este princípio (0-1-2-3-**4-5-4-7**-4-8, 0-1-2-3-**4-5-4-6-4-7**-4-8, ...) sempre aumentando o número de iterações do *loop*.

A.4.1 Cobertura Média dos Algoritmos

Neste SUT o número de avaliações, assim como no SUT busca linear, foi limitado a 400.000. Na figura A.19 percebe-se que apenas a geração aleatória, o algoritmo GEOcan e o GEOreal foram capazes de atingir a cobertura de 100% antes do limite de 400.000 avaliações. Destes, o GEOreal foi o primeiro a atingir esta cobertura próximo da avaliação 75.000. Os algoritmos SGA, RCGA não cobriram qualquer caminho.

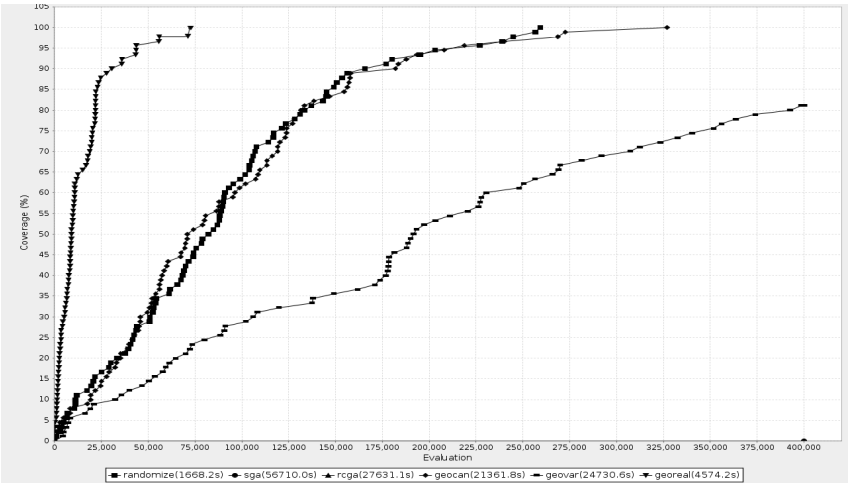


Figura A.19: SUT Busca Binária (400.000 avaliações) – cobertura média dos algoritmos usando FO *Similarity*

Os resultados utilizando a FO Bueno e Jino (figura A.20) foram muito parecidas com as utilizando a FO *Similarity*. O GEOreal obteve o melhor desempenho enquanto que os algoritmos GEOcan, GEOvar e geração aleatória atingiram boa cobertura.

A.4.2 Avaliação da Função Objetivo *Similarity*

A figura A.21 mostra a avaliação da FO deste SUT plotado no domínio total da única variável de entrada [0,16838]. A utilização da escala logarítmica neste gráfico nos permite

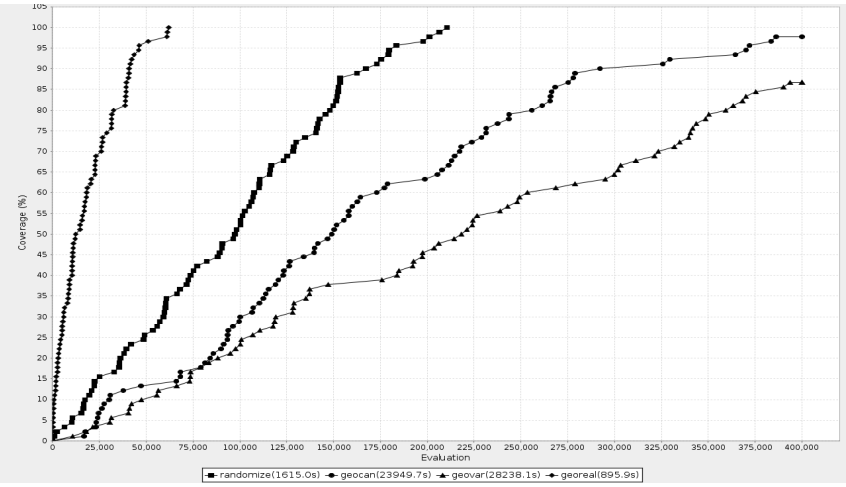


Figura A.20: SUT Busca Binária (400.000 avaliações) – cobertura média dos algoritmos usando FO Bueno e Jino

visualizar doze níveis de pontos. Destes, seis são responsáveis por todos os pontos (X,Y) que possuem o valor X como sendo valores que estavam no vetor de busca. Esses são os níveis (contados de baixo para cima) três, quatro, seis, sete, dez, onze e doze. Os outros seis níveis possuem todos os outros pontos (X,Y) que possuem o valor X fora do vetor de busca. O que diferencia todos esses níveis é o número de iterações e os condicionais executados no interior do *loop*. Esses condicionais acabam por concatenar diferentes *chars* na *string* do caminho o que faz com que a FO avalie-os com valores diferentes.

Fica claro neste gráfico que a FO é incapaz de guiar uma busca pelo valor alvo. Todos

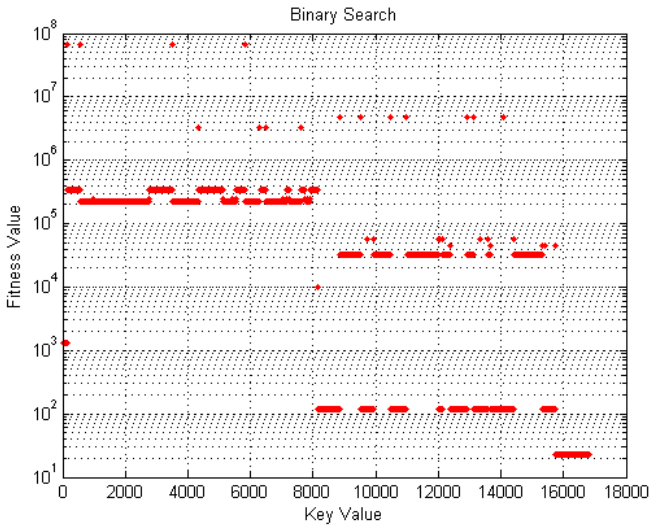


Figura A.21: Avaliação da FO *Similarity* - Busca Binária

os pontos do domínio estão espalhados por diversos níveis sem que haja qualquer ligação funcional entre eles de forma que uma metaheurística possa procurar por um máximo ou mínimo global.

A.4.3 Avaliação da Função Objetivo Bueno e Jino

A avaliação da FO Bueno e Jino A.22 mostra um ponto com grande valor de avaliação e três retas. O ponto com $X \sim 4000$ representa o valor de entrada que exercita o caminho alvo e portanto é avaliado com o maior valor possível pela FO. Para compreender as retas é necessário compreender o funcionamento do SUT Busca Binária. No código do SUT temos um *loop* que executa a busca pelo valor entrado X em um vetor com 40 valores ordenados (gerados aleatoriamente). O valor de entrada X é buscado iniciando-se pelo valor que está no meio do vetor (posição 20). Caso o valor de X seja menor que o da posição corrente, verifica-se o valor na metade inferior (posição 10). Caso o valor de X seja maior, verifica-se o valor na metade superior (posição 30).

Este entendimento é essencial para compreender porque as três retas mostradas no

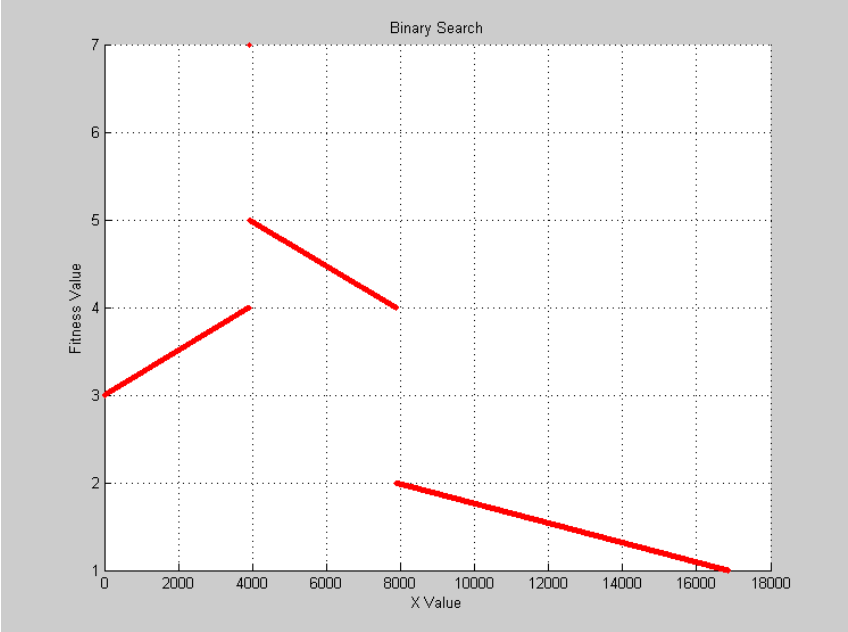


Figura A.22: Avaliação da FO Bueno e Jino - Busca Binária

gráfico de avaliação da FO estão separadas. O caminho alvo escolhido foi o caminho que leva o SUT a encontrar o valor de X na posição 10, ou seja, a busca verifica a posição 20 e depois move para a metade inferior (posição 10) e encontra o valor buscado. Todos os

valores menores que o da posição 10 desviam-se do caminho alvo possuindo quatro arestas em comum. Portanto esses valores de X são penalizados conforme X se afasta de 4000. Esta é a reta mostrada entre os valores $0 < X < 4000$.

Para os valores de entrada X que são maiores que o valor na posição 10 e menores que a posição 20 (primeira posição checada pelo SUT) um caminho diferente é percorrido (do que o percorrido pelos valores entre 0 e o valor da posição 20). Este caminho possui cinco arestas em comum com o caminho alvo. Portanto esses valores de X são avaliados pela FO com valor cinco e penalizados conforme X se afasta de 4000. Esta é a segunda reta (reta do meio na figura A.22).

Finalmente existem os valores de X maiores que o valor na posição do meio (posição 20). Estes exercitam outro caminho que apenas possui duas arestas iguais ao do caminho alvo (por isto o valor no início da reta vale dois). A partir dai a penalização aumenta conforme o valor de X caminha para 16383.

A.5 Valor do Meio

O SUT Valor do Meio possui três variáveis de entrada. Seu código interno apenas possui condicionais que verificam qual dos três valores entrados é o valor do meio. Por este motivo este SUT é muito parecido com o SUT Triângulo Simplificado mas possui uma complexidade ciclomática menor.

A partir do GFC (figura A.23) os seis caminhos independentes foram escolhidos para serem cobertos. São eles 0-1-2-3-9, 0-1-2-3-4-9, 0-1-2-3-5-9, 0-1-2-6-9, 0-1-2-6-7-9 e 0-1-2-6-8-9.

A.5.1 Cobertura Média dos Algoritmos

Neste SUT o número de avaliações da FO foi limitado a apenas 15.000. A partir da figura A.24 observa-se que com apenas 1.000 avaliações quase todos os algoritmos foram capazes de alcançar 100% de cobertura – apenas o GEOcan ainda não havia atingido esta marca. Apesar disto, ele atinge total cobertura dos caminhos próximo de 10.500 avaliações.

Apesar da geração aleatória obter a melhor eficiência os algoritmos GEOvar, GEOreal, SGA e RCGA têm desempenhos muito parecidas com o primeiro colocado. Nestes dois últimos, vemos que o desempenho foi exatamente igual (todos os pontos estão sobrepostos). Observa-se também que ambos descrevem uma reta do 0% a 100% de cobertura. Isso ocorre pois os algoritmos SGA e RCGA só verificam se a solução procurada pelo algoritmo foi encontrada quando toda a população é evoluída através dos operadores

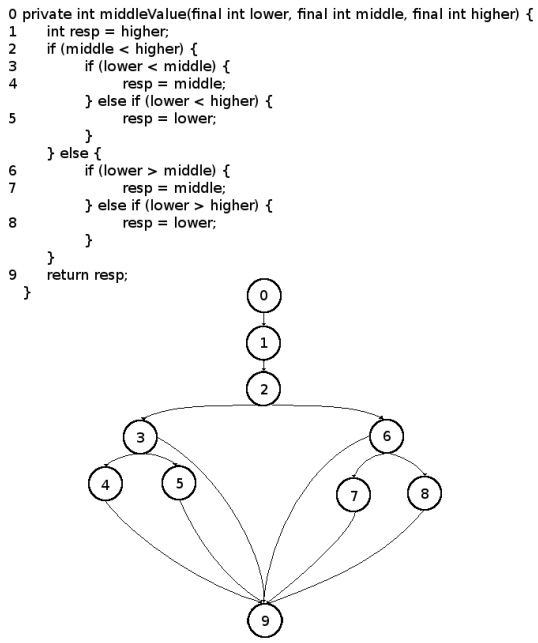


Figura A.23: CFG - Valor do Meio

genéticos. Neste SUT a população dos algoritmos SGA e RCGA é de cem indivíduos e por este SUT ser simples o algoritmo encontra cada uma das seis soluções (uma para cada caminho a ser coberto) em cada evolução da população. Isto ocorre em todas as vinte execuções do algoritmo. Ao fazermos a média da cobertura, a curva apresenta a característica de reta bem demarcada.

O desempenho dos algoritmos utilizando a FO Bueno e Jino pode ser observada com 13.000 avaliações na figura A.25. Aqui também vemos que a geração aleatória obteve o melhor desempenho sobre os outros algoritmos mas de forma geral, quando comparando os desempenhos da FO *Similarity*, vemos uma pequena melhora.

A.5.2 Avaliação da Função Objetivo *Similarity*

Dentre todos os SUTs este é o único que possui em seu domínio valores negativos. Por este motivo ao plotar o gráfio de avaliação da função objetivo o domínio foi restringido a [-50,50]. Das três variáveis, a terceira foi fixada com valor zero que é exatamente o meio do domínio.

Da figura A.26 percebe-se claramente seis planos sendo dois deles retângulos e quatro sendo triângulos. Da mesma maneira, neste SUT temos seis caminhos a serem cobertos. Sabemos que a primeira variável de entrada é o eixo X, a segunda variável de entrada é

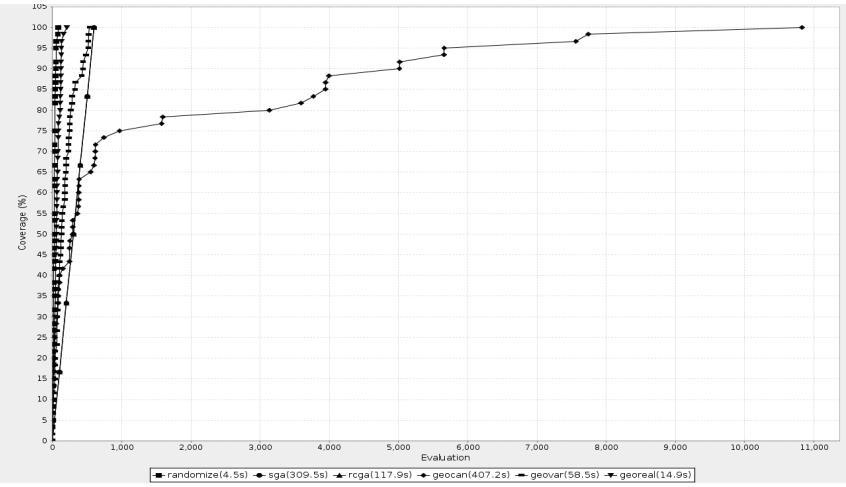


Figura A.24: SUT Valor do Meio (11.000 avaliações) – cobertura média dos algoritmos usando FO *Similarity*

o eixo Y e a terceira variável é sempre zero. Analisando o gráfico, temos no plano mais alto os conjuntos de pontos (X,Y,0) em que os valores de X são sempre positivos e os de Y são sempre negativos. Neste planos fica bem definido o caminho que aponta como valor do meio a terceira variável de entrada com valor zero. O mesmo acontece no outro plano que é um retângulo. Neste caso os valores de X são sempre negativos e os de Y positivos levando a terceira variável de entrada ser o valor do meio. Até aqui já temos dois caminhos cobertos.

Sobraram os quatro planos com formato de triângulo. Os dois mais baixos estão na região onde os valores de X e Y são positivos. Portanto dependemos dos valores das duas primeiras variáveis de entrada para sabermos qual é o valor do meio – e consequentemente qual caminho será coberto. O caminho que indica a primeira variável como o valor do meio ocorre nos conjuntos de pontos (X,Y,0) onde os valores de X são maiores que Y (plano mais inferior). Quando o contrário acontece ($Y > X$) temos o caminho que aponta a segunda variável de entrada como o valor do meio (segundo plano inferior). Todo este raciocínio é análogo para os outros dois planos triangulares só que neste caso os valores de X e Y são ambos negativos.

A.5.3 Avaliação da Função Objetivo Bueno e Jino

A partir das figuras A.27 e A.28 observamos os mesmos seis planos vistos em A.26. São os conjuntos de pontos (X,Y,0) que exercitam cada um dos seis caminhos deste SUT. A diferença aqui é que estes planos estão inclinados, excetuando-se o mais alto. Este mostra o conjunto de pontos de executam o caminho alvo e por isso foram avaliados com maior

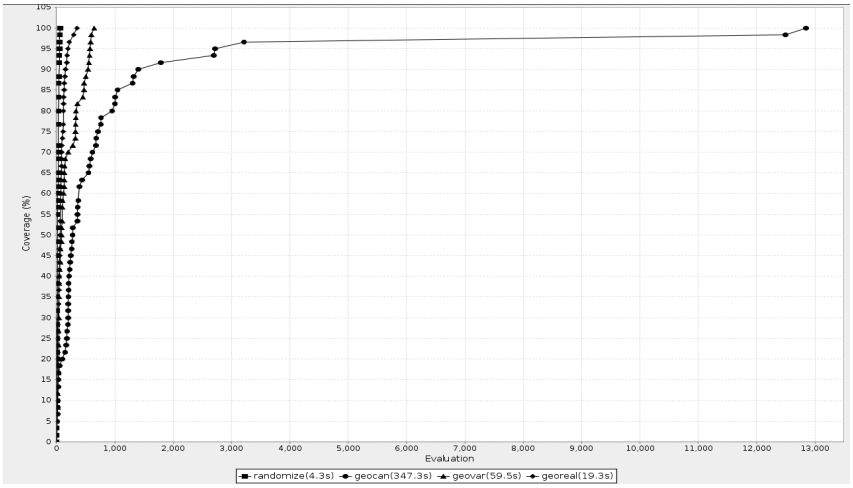


Figura A.25: SUT Valor do Meio (13.000 avaliações) – cobertura média dos algoritmos usando FO Bueno e Jino

valor pela FO.

O fato dos plano terem certa inclinação é um diferencial quando comparado com a avaliação da FO *Similarity*. Isso ajuda os algoritmos baseados nesta FO a terem um direcionamento na busca pela solução com maior avaliação.

A.6 Triângulo

O SUT Triângulo fica atrás apenas do SUT Triângulo Simplificado em complexidade ciclomática. O primeiro é uma derivação mais completa deste último sendo que agora o triângulo é classificado em seis categorias: "não é triângulo", equilátero, isósceles, reto, acutângulo e obtusângulo. O código interno deste SUT é composto apenas de condicionais que verificam em qual categoria os três valores de entrada (que são os lados do triângulo) coloca o triângulo formado por eles. Como temos seis categorias, seis são os caminhos independentes. São eles (a partir da figura A.29): 0-1-2-3-13, 0-1-2-4-5-6-13, 0-1-2-4-5-7-13, 0-1-2-4-8-9-10-13, 0-1-2-4-8-9-11-13 e 0-1-2-4-8-9-12-13.

A.6.1 Cobertura Média dos Algoritmos

O número de avaliações da FO foi limitado a 100.000 pois a partir deste número não há grande mudança no desempenho dos algoritmos. Podemos perceber na figura A.30 que todos os algoritmos não foram capazes de atingir a cobertura total. Os seis caminhos independentes foram ordenados em grau de dificuldade crescente de cobertura (iniciando

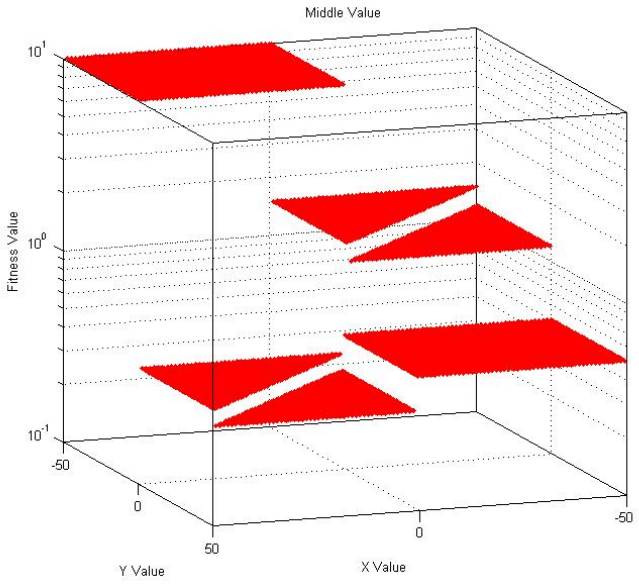


Figura A.26: Avaliação da FO *Similarity* - Valor do Meio

com "não triângulo", acutângulo, obtusângulo, isósceles, reto e equilátero). De fato, existem muito mais pontos no domínio (X,Y,Z) que cobrem o caminho "não triângulo" e por este motivo ele é o primeiro da lista de caminhos a serem cobertos. Essa lógica é seguida até o sexto e último caminho que é o mais difícil de ser coberto: o triângulo equilátero.

Sendo assim os últimos dois caminhos da lista (67% de cobertura) acabam por serem cobertos apenas pelo GEOreal e RCGA. Já os algoritmos SGA, GEOcan, GEOvar e geração aleatória que não passaram dos 22.5%. Eles foram incapazes de encontrar em todas as execuções os caminhos para o triângulo isósceles, reto e equilátero. Por este motivo fica perceptível que apenas os algoritmos baseados em codificação real conseguem bom desempenho neste SUT.

A partir da figura A.31 observa-se a cobertura para a FO Bueno e Jino. Percebe-se o mesmo comportamento observado na FO *Similarity*. O algoritmo GEOreal rapidamente atinge 67% enquanto que os outros algoritmos estagnam próximo de 17%.

A.6.2 Avaliação da Função Objetivo *Similarity*

Este SUT possui três variáveis de entrada mas apenas duas foram variadas no domínio entre [0,100]. A terceira variável de entrada foi fixada com o valor doze. O resultado pode

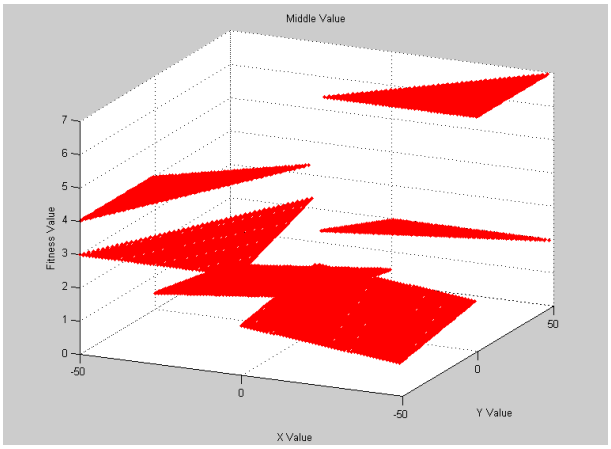


Figura A.27: Avaliação da FO Bueno e Jino - Valor do Meio

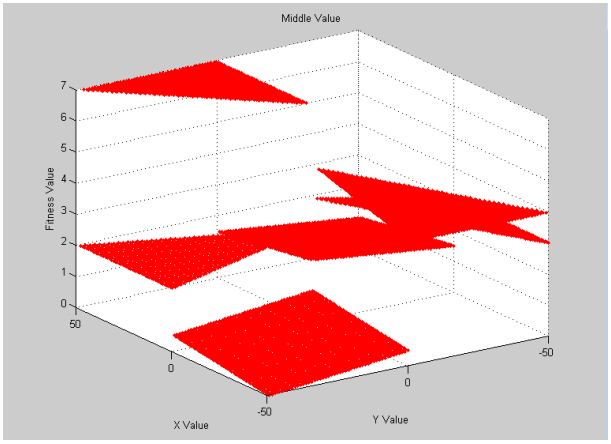


Figura A.28: Avaliação da FO Bueno e Jino - Valor do Meio

ser observado nas figuras A.32 e A.33

Existem cinco planos nestes gráficos. O primeiro é o plano mais inferior e com o maior número de pontos do domínio. Estes são os conjuntos de pontos (X,Y,12) que conduzem ao caminho do "não triângulo". São pontos onde $x+y > z$ ou $x+z > y$ ou $y+z > x$. Em seguida temos um plano formado de três retas. Essas retas são formadas pelos pontos que conduzem o fluxo da execução do SUT pelo caminho do triângulo isósceles. As três retas são aquelas onde $X = 12$, $Y = 12$ e $X = Y$. Mais acima temos o terceiro plano formado pelos pontos que conduzem ao caminho do triângulo acutângulo. Ele possui os pontos onde o triângulo formado tem entre os catetos (Y e Z) um ângulo maior que noventa graus. O plano seguinte (com formato de uma ponta de faca) são dos pontos que conduzem o SUT pelo caminho do triângulo obtusângulo. Finalmente, ao topo temos quatro pontos que executam o caminho do triângulo reto. Eis aqui o motivo pelo qual a terceira variável

```
0 private TriangleType complexTriangle(final int a, final int b, final int c) {
1     TriangleType type;
1     int d;
2     if ((a > b + c) || (b > c + a) || (c > a + b)) {
3           type = TriangleType.NO_TRIANGLE;
      } else {
4           if ((a == b) || (b == c) || (a == c)) {
5                 if ((a == b) && (b == c)) {
6                       type = TriangleType.EQUILATERO;
                  } else {
7                       type = TriangleType.ISOSCELES;
                  }
            } else {
8                 final int aa = a * a;
8                 final int bb = b * b;
8                 final int cc = c * c;
8                 d = bb + cc;
9                 if (aa == d) {
10                      type = TriangleType.RETO;
                  } else if (aa < d) {
11                      type = TriangleType.ACUTANGULO;
                  } else {
12                      type = TriangleType.OBTUSANGULO;
                  }
            }
      }
13    return type;
  }
```

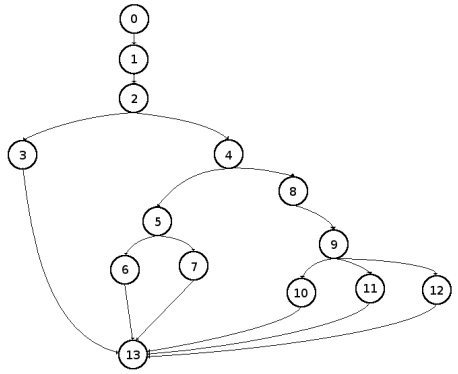


Figura A.29: CFG - Triângulo

de entrada foi escolhida com valor doze. Com este valor podemos encontrar no domínio de [0,100] quatro conjuntos (X,Y,12) que formam um triângulo retângulo. São eles (37,35,12), (20,16,12), (15,9,12) e (13,5,12). Finalmente temos o último ponto que conduz à execução do SUT pelo caminho do triângulo equilátero. Este ponto não está visível nos gráficos e ele é formado pelos valores (12,12,12).

A.6.3 Avaliação da Função Objetivo Bueno e Jino

O domínio das variáveis X e Y foram limitadas a [0,100] sendo que a variável Z foi fixada com valor 50. Sendo assim vemos nas figuras A.34 e A.35 todos os pontos (X,Y,50) que executam o SUT tentando exercitar o caminho alvo do triângulo reto – caminho 0-1-2-4-8-9-10-13 na figura A.29.

Observa-se três grandes conjuntos de pontos. O conjunto mais baixo, formado por três triângulos inclinados, são os pontos que exercitam o caminho "não triângulo" e que possuem apenas três arestas em comum com o caminho alvo (**0-1-2**-3-13).

Em seguida vemos ao meio três retas inclinadas. Estes são os pontos (X,Y,50) que exercitam os caminhos do triângulo isósceles e equilátero. São os caminhos que possuem

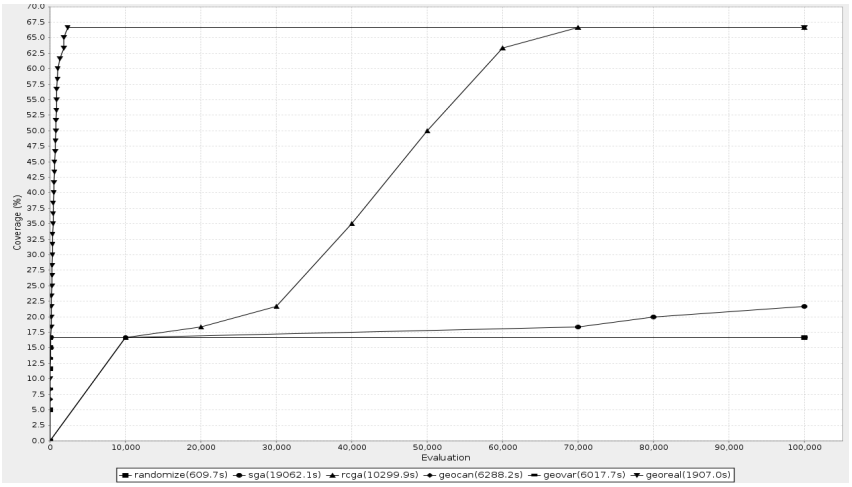


Figura A.30: SUT Triângulo (100.000 avaliações) – cobertura média dos algoritmos usando FO *Similarity*

quatro arestas em comum com o caminho alvo: **0-1-2-4**-5-6-13 e **0-1-2-4**-5-7-13.

Por fim temos mais ao alto todos os pontos que exercitam os caminhos restantes. Estes são para os triângulos acutângulo, obtusângulo e reto (alvo). Os dois primeiros desviam do caminho alvo após seis arestas em comum: **0-1-2-4-8-9**-11-13 e **0-1-2-4-8-9**-12-13. Por estes serem os caminhos que mais possuem arestas em comum com o caminho alvo, são avaliados com alto valor pela FO mas possuem uma penalidade. Por este motivo vemos essa curva no topo das figuras A.34 e A.35.

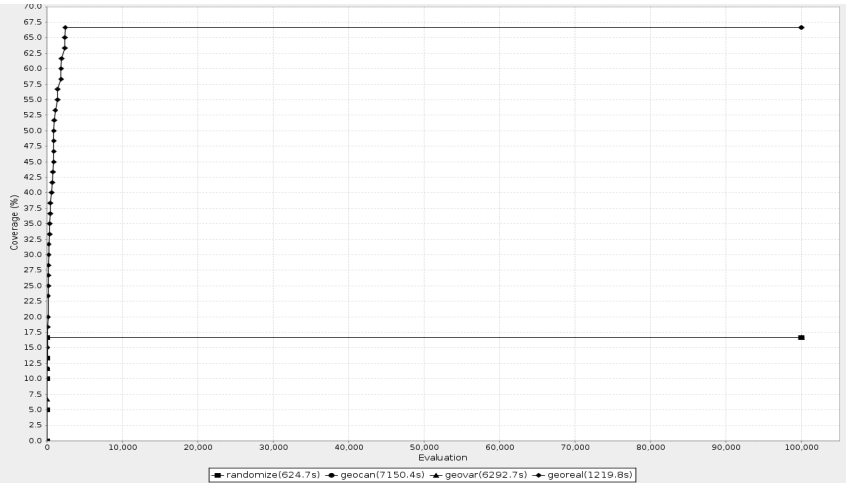


Figura A.31: SUT Triângulo (1.400 avaliações) – cobertura média dos algoritmos usando FO Bueno e Jino

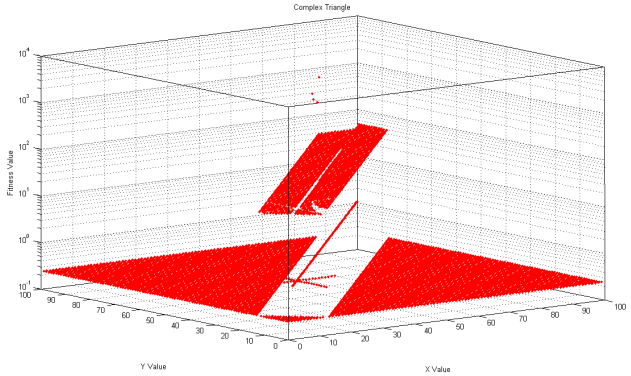


Figura A.32: Avaliação da FO *Similarity* - Triângulo

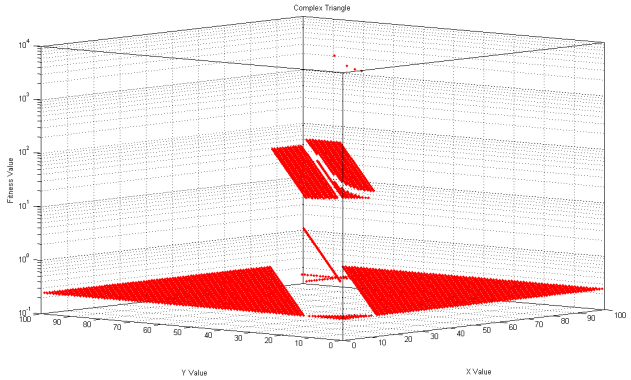


Figura A.33: Avaliação da FO *Similarity* - Triângulo

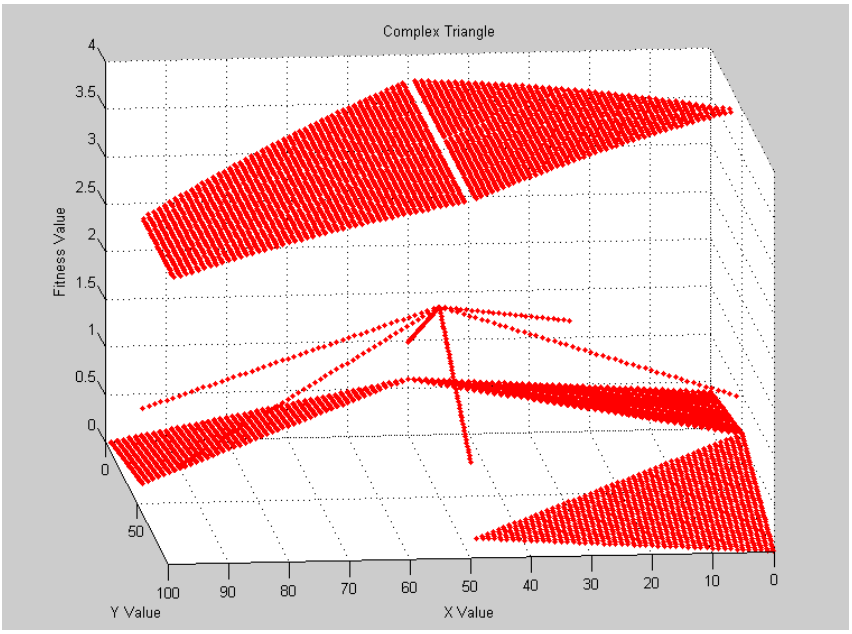


Figura A.34: Avaliação da FO Bueno e Jino - Triângulo

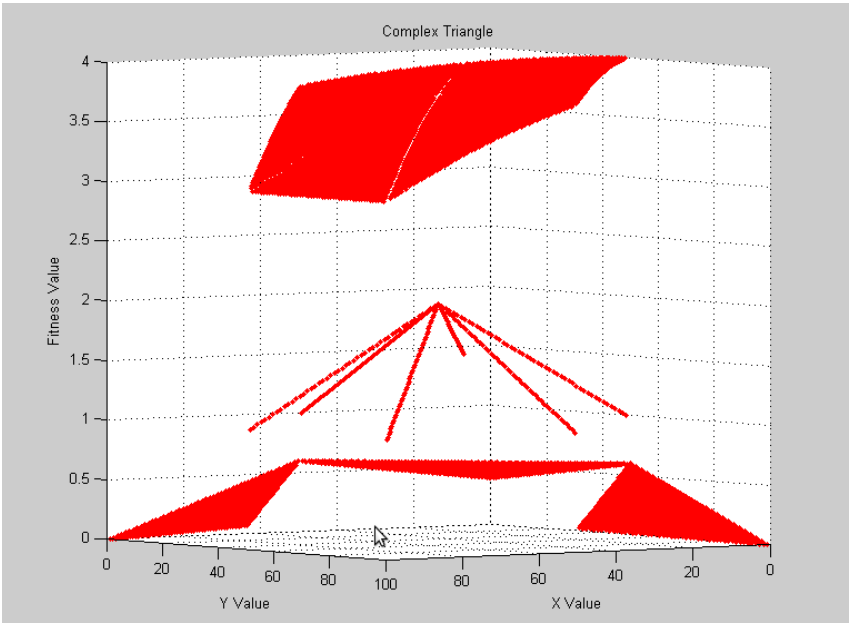


Figura A.35: Avaliação da FO Bueno e Jino - Triângulo

Apêndice B

Detalhamento Técnico dos Algoritmos SGA e RCGA

Este capítulo do apêndice tem por objetivo detalhar a implementação dos algoritmos SGA e RCGA. Durante todo o texto apresentado neste trabalho muitos detalhes técnicos da linguagem Java foram propositalmente omitidos para não atrapalhar a compreensão daqueles leitores não acostumados com esta área. Abaixo são apresentadas informações relevantes para aqueles que desejarem replicar a implementação dos algoritmos SGA e RCGA utilizados no estudo empírico e desta maneira validar os resultados apresentados.

Ambos os algoritmos foram implementados com a ajuda do pacote JGAP[1]. Eis os detalhes das classes utilizadas para o SGA:

```
1  Configuration c = new DefaultConfiguration();
2  c.setKeepPopulationSizeConstant(true);
3  MutationOperator mutation = new MutationOperator(c, <probabilidade-mutacao>);
4  c.addGeneticOperator(mutation);
5  CrossoverOperator crossover = new CrossoverOperator(c,<probabilidade-crossover>);
6  c.addGeneticOperator(crossover);
7  Gene[] sampleGenes = new Gene[variables];
8  for (int i = 0; i < variables; i++) {
9    sampleGenes[i] = new FixedBinaryGene(c, <numero-de-bits-por-variavel>);
10 }
11 IChromosome sampleChromosome = new Chromosome(c, sampleGenes);
12 Genotype population = Genotype.randomInitialGenotype(c);
```

Para o RCGA basta trocar a linha 9 por:

```
sampleGenes[i] = new DoubleGene(c,<limite-dominio-inferior>, <limite-dominio-superior>);
```

[1] http://jgap.sourceforge.net/